Aceleración de algoritmos de estimación de movimiento mediante OpenCL

David Díaz Morgado y David Gómez Blanco

GRADO EN INGENIERÍA INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTESNE DE MADRID

Trabajo Fin de Grado en Ingeniería Informática

17 de Junio de 2016

Director/es y/o colaborador:

Carlos García Sánchez
Guillermo Botella

Autorización de difusión

David Díaz Morgado y David Gómez Blanco

17 de Junio de 2016

Los abajo firmantes, matriculados en el Grado en Ingeniería Informática de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores el presente Trabajo Fin de Grado: "Aceleración de algoritmos de estimación de movimiento mediante OpenCL", realizado durante el curso académico 2015-2016 bajo la dirección de Carlos García Sánchez y Guillermo Botella en el Departamento de Arquitectura de Computadores, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Dedicatoria

Dedicado a mi familia, amigos, compañeros y profesores, sin su ayuda no estaría aquí.

- David Díaz Morgado.

A todas esas personas que me preguntaron:

¿De que trata tu TFG?

y yo les respondí:

Déjalo... No lo vas a entender

- David Gómez Blanco

Agradecimientos

Antes de nada, agradecer a Guillermo y a Carlos por todo su tiempo y paciencia. Estamos convencidos que sin su ayuda no hubiéramos sido capaces de abordarlo todo. Gracias también por sus lecciones y por sus ánimos en todo momento.

Quería agradecer todo esto a mucha gente.
En primer lugar a mi familia la cual ha estado apoyando durante todo este camino y siempre tuvieron confianza en mi criterio y mi forma de hacer las cosas. A mi perro por ser tan puta cabra y hacerme reír tantas veces. A mis amigos y compañeros por todas esas momentos buenos y risas tanto en la facultad como fuera. A mis profesores de primaria, E.S.O. y bachillerato por enseñarme la base para poder ser todo lo que soy hoy. A los de la facultad por transformarme de nadie a ingeniero.
Muchas gracias a todos.

David Diaz Morgado.

Agradecer a tantas personas que poner sus nombres abarcaría otro trabajo nuevo.
A todas esas personas importantes para mi: Al pápa, a la máma, al tate y a toda mi familia, que no hay recuerdo que recuerde sin ellos. A los shurmachos y no tan shurmachos, que van y vienen pero nunca se detienen. A los compañeros de facultad, con los que coger mesa en la cafetería suponía ocupar una mesa grande y dos sillas más. A los radonenses, que era poco lo que nos veíamos, pero lo poco era bueno. Y agradecer a los que están y sobre todo a los que ya no están.
Y por último agradecer a Violeta... agradecer a Violeta por todo lo que me ha aguantado y le queda por aguantar.

David Gómez Blanco

Resumen en castellano

El flujo óptico y la estimación de movimiento es área de conococimiento muy importante usado en otros campos del conocimiento como el de la seguridad o el de la bioinformática. En estos sectores, se demandan aplicaciones de flujo óptico que realicen actividades muy importantes con tiempos de ejecución lo más bajos posibles, llegando a tiempo real si es posible. Debido a la gran complejidad de cálculos que siguen a este tipo de algoritmos como se observará en la sección de resultados, la aceleración de estos es una parte vital para dar soporte y conseguir ese tiempo real tan buscado. Por lo que planteamos como objetivo para este TFG la aceleración de este tipo de algoritmos mediante diversos tipos de aceleradores usando **OpenCL** y de paso demostrar que **OpenCL** es una buena herramienta que permite códigos paralelizados con un gran *Speedup* a la par que funcionar en toda una diversa gama de dispositivos tan distintos como un *GPU* y una *FPGA*. Para lo anteriormente mencionado trataremos de desarrollar un código para cada algoritmo y optimizarlo de forma no especifica a una plataforma para posteriormente ejecutarlo sobre las diversas plataformas y medir tiempos y error para cada algoritmo. Para el desarrollo de este proyecto partimos de la teoría de dos algoritmos ya existentes: **Lucas&Kanade** monoescala y el **Horn&Schunck**. Además, usaremos estímulos para estos algoritmos muy aceptados por la comunidad como pueden ser el **RubberWhale** o los **Grove**, los cuales nos ayudarán a establecer la corrección de estos algoritmos y analizar su precisión, dando así un estudio referencia para saber cual escoger.

Palabras clave

Flujo óptico, seguridad, bioinformática, OpenCL, Paralelismo, Speedup, aceleradores, Lucas&Kanade, Horn&Schunck, estímulo.

Abstract

Optical flow is a very important field used in other sectors such as security or bioinformatics. They demand optical flow applications that make really important activities with the smallest times, reaching real time processing. Due to algorithms complexity, accelerating is crucial to achieve real time constrains. In addition, we analyze the behavior of algorithms under study in different kind of accelerators using OpenCL in order to demonstrate that OpenCL provides an excellent trade-off between performance rate and coding effort. We would like to remark that OpenCL is already supported by most of GPU and multicore vendors including recently FPGAs. In order to complete a full study, we have developed a non-optimized source for each algorithm without specific platforms optimizations, to easy evaluate times and accuracy error for each algorithm. In this project we have focus on Lucas&Kanade and Horn&Schunck's multi-scale which are widely accepted by the community. As input stimuli we have used the well-known **RubberWhale** or **Groves**, in order to establish the correction and analyze its accuracy, giving us a study which help switching between the algorithms for applications which make optical flow.

Keywords

Optical flow, security, bioinformatics, OpenCL, parallelism, Speedup, accelerators, Luka&Kanade, Horn&Schunck, stimulus.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

La cantidad de información que necesitamos procesar crece exponencialmente con el paso del tiempo, es por esto que cada vez más necesitemos contar más con máquinas que favorezcan el paralelismo a nivel de datos a un bajo coste computacional. A partir de esta idea nacieron arquitecturas *multicore* y *manycore* usadas en elementos *hardware* como procesadores y unidades de procesamiento gráfico o *GPUs*, prometiendo un alto rendimiento con un bajo consumo y a un coste bastante inferior al que nos venían acostumbrando.

El reto esta vez es encontrar un lenguaje de programación que no discrimine entre diferentes arquitecturas. De esta necesidad nace **OpenCL**, y será en este lenguaje en el que nos enfocaremos por ser este una alternativa libre y compatible con la mayoría de procesadores en el mercado lo que, sumado al apoyo de compañías como ***Apple***, ***NVidia*** y ***ATI***, pone de relieve el futuro tan prometedor que tiene.

1.1. Motivación

Sistemas empotrados como pueden ser las *Smart Glasses* (en las cuales ***Google*** está trabajando), pasando por pequeños *drones*, hasta llegar a vehículos de conducción propia como son los *Smart Cars*; sistemas de reconocimiento de patrones aplicados a la salud (como la detección del cáncer de mama) o cámaras de seguridad de alta precisión son solo algunas de las aplicaciones que tienen en común el uso de los algoritmos de estimación de movi-

miento. Es necesario, por tanto, que toda esa información sea tratada de manera amigable y eficiente intentando que, viendo la tamaña cantidad de productos que hay actualmente en el mercado, pueda ser funcional sobre el máximo de plataformas posibles, de esta manera, queda abierto a una fácil portabilidad.

Bajo estas premisas, con este trabajo queremos converger hacia el estudio de un modelo que facilite a los ingenieros la tarea de programar para distintas plataformas que ofrezcan características de rendimiento y consumo muy diversas. Queremos acercar **OpenCL** al lector para que, de esta manera, estemos un paso más de cerca de conseguir un método heterogéneo de programación aplicado a uno de los temas más importantes de la I.A. como es, en este caso, la visión por ordenador[1].

Queremos, por tanto, presentar **OpenCL** como la primera opción a escoger cuando se trata de aceleración de algoritmos, situándolo para el caso de este trabajo en el marco de la visión por ordenador y, más concretamente, en la estimación del movimiento.

1.2. Estimación de movimiento y flujo óptico

La percepción computacional es una rama de la computación centrada en traducir todas las funciones perceptoras de un organismo vivo a lenguaje máquina, para lo cual se necesita una cantidad de información desmesurada que esté recibiendo de manera continuada.

La visión es una capacidad principal de muchos organismos vivos. Permite captar de forma continua mucha cantidad de información en forma de colores y formas, lo que nos permite hacernos una idea del entorno que nos rodea y generando respuestas al mismo.

Una de muchas operaciones que se hace con esa información es la de estimar el movimien-

[1]Es una disciplina cientifica que incluye métodos para adquirir y procesar información a través de las imágenes que ofrece el mundo real de la misma manera que hace nuestro cerebro en unión con los ojos.

to de los objetos de la escena, cuyo objetivo es el de generar un campo de vectores espaciales que representan el movimiento de todos los elementos entre dos instantes de tiempo o *frames* de manera que, leyendo estos vectores, seamos capaces de obtener un flujo óptico con el que estimar el movimiento y, de esta manera, predecirlo para generar una respuesta lo antes posible.

1.3. Clasificación de algoritmos

Dentro de las técnicas existentes de estimación de movimiento y flujo óptico podemos enumerar tres tipos de algoritmos que elegiremos dependiendo de la de la aplicación que se les quiera dar, teniendo en cuenta cuál se ajusta mejor al *problema de la correspondencia*[2], formado este, a su vez, del problema de *aliasing*[3] y de apertura[3].

1.3.1. Modelos de energía

Consiste en utilizar filtros que responden de manera distinta y, para muchos casos, óptima a ciertas velocidades. Para que esto ocurra son necesarios varios filtros en paralelo que se irán activando para ciertos rangos de valores. En esta técnica, es común el uso de filtros basados en la **Transformada de Fourier** para calcular el flujo a través del dominio de la frecuencia.

La estimación del movimiento es planteada como un problema *bayesiano*, en el cual buscamos maximizar la probabilidad del movimiento observando la intensidad de la siguiente trama. Se usan dos funciones de densidad de probabilidad: Una es la probabilidad condicional de la intensidad de la imagen a partir del campo de movimiento, y la otra es la probabilidad de los propios vectores de movimiento.

[2]La correspondencia consiste en reconocer el mismo elemento de un flujo óptico entre dos instantes de tiempo distintos lo que supone que, en una película de varios *frames*, se tenga conocimiento de la posición de un mismo elemento a través de todos los *frames* (completando de esta manera todas las tuplas de la forma (x, y, n), siendo n cada uno de los *frames*).

[3]El **aliasing** es el efecto que causa que señales continuas distintas se tornen indistinguibles cuando se muestrean digitalmente. Cuando esto sucede, la señal original no puede ser reconstruida de forma unívoca a partir de la señal digital

Como inconveniente tenemos el tiempo necesario para realizar los cálculos ya que, además de realizar el procesamiento *pixel* a *pixel*, se incluye una distribución de sus velocidades que servirá para ejecutar un filtro u otro, que puede ser más o menos complejo.

Este modelo se asemeja mucho a un modelo que presentaremos más tarde: el modelo del gradiente. Ambos utilizan filtros espaciales aunque el de energía los use para orientaciones espacio-temporales especificas y los de gradiente para calcular el cociente con los filtros.

1.3.2. Modelos de *matching*

Como su propio nombre indica, este método se basa en comparar la misma región de *pixeles* entre dos imágenes o *frames* para estudiar los cambios acontecidos en el mismo. Estas regiones son apodadas **macrobloques**. Mediante el estudio de estos macrobloques se puede estimar el flujo entre las dos imágenes y, además, inferir la velocidad del movimiento. En algunos casos se usan ventanas dinámicas, que se desplazan por la imagen estudiando el flujo entre sus límites.

Los criterios de búsqueda de semejanza entre bloques son: el de *Sumatorio de Diferencias Absolutas (**SAD**)*, *Correlación Cruzada Normalizada (**NCC**)* y *Clasificación por Diferencia de Pixel (**PDC**)*. Todos estos criterios se encargan de buscar el bloque que más se parezca al conjunto de bloques del frame directamente anterior. Si durante esta búsqueda encontramos una coincidencia y está desplazado del original, entonces podemos asegurar que ha habido movimiento. En este caso, se establecerá una velocidad a cada uno de los *pixeles*, que será la misma para todos.

Un inconveniente a estos algoritmos es la lentitud en el procesamiento de los datos. Supongamos una imagen de P *pixeles*, un bloque de B *pixeles* y una ventana de tamaño V.

En el caso peor, nuestro algoritmo tendrá un coste de $P * B * V$. Se ha intentado minimizar este computo mediante diversas técnicas como **TSST**[12] (*Three Step Search Technique*) o **LOGST**[14](*2D Logarithm Search Technique*).

Otra cosa a tener en cuenta es la dualidad entre precisión y *performance* (*accuracy-efficency*), ya que a mayor tamaño de ventana mayor precisión, pero también más computo.

1.3.3. Modelos de gradiente

Estos modelos basan su comportamiento en aplicar derivadas espacio-temporales sobre las intensidades de los *pixeles* de la imagen para obtener los vectores de velocidad que forman el flujo óptico. A las derivadas que son aplicadas para las intensidades de los *pixeles* se las llama filtros y pueden variar en tamaño. En general, este modelo sugiere una buena estimación.

Tiene un enfoque distinto a los mencionados anteriormente. Estos últimos básicamente hacen uso de plantillas para obtener un ajuste entre la plantilla y el movimiento. Lo malo de esta práctica es que queda muy a merced del contraste, lo que implica añadir una etapa más para una normalización bastante costosa. También sería necesario añadir una etapa más, pues con el *matching* entre plantilla-movimiento no hallamos la velocidad.

Por lo tanto, vemos como los modelos de gradiente tienen menor número de etapas, ya que las velocidades que queremos hallar pueden ser calculadas simplemente como cociente de las derivadas (o filtros) para cada *pixel* de la imagen. Esto tiene como ventaja que no es necesaria otra etapa más para normalizar el contraste en las imágenes, ya que este varía de la misma forma tanto para el denominador como para el numerador del cociente, por lo cuál acaba anulándose.

Como único inconveniente encontramos que es necesario invertir grandes cantidades de

información en forma de matrices para los filtros de las derivadas. Dos grandes ejemplos a estos modelos son los que encontraremos en este documento: ***Lucas&Kanade*** [15] y ***Horn&Schunck*** [11].

1.4. Objetivo y organización

A continuación se enumera el contenido del resto de la memoria del Trabajo Fin de Grado.

En el capítulo 2 encontramos la historia de las unidades de aceleramiento y el estado del arte, seguidos de los métodos de programación, divididos en programación por medio de directivas y programación dedicada de *GPUs*, en la que destacaremos a **OpenCL**.

En el capítulo 3 nos centraremos en los algoritmos que usaremos en nuestro proyecto, así como de las métricas que usaremos para establecer como de buena es nuestra implementación y de los estímulos que tomarán como entrada nuestros programas.

Para el capítulo 4 abordaremos la implementación de los algoritmos mediante el estandard OpenCL. A continuación, en el capítulo 5 expondremos al público los resultados obtenidos en los que se incluye resultados gráficos, tablas y comentarios.

Para finalizar, en el capítulo 6, se reservará un apartado de conclusiones y se presentarán los posibles trabajos futuros.

Capítulo 2

Aceleradores y su paradigma de programación

2.1. Unidades de aceleramiento de procesamiento

2.1.1. Historia de los aceleradores

Las primeras tarjetas gráficas que empezaron a desarrollarse datan de los años 60. Estas unidades de procesamiento permitieron visualizar gráficos en unos monitores que, hasta la fecha, solo podían mostrar texto. Con la aparición de este nuevo *hardware*, se extendió su uso en detrimento de las impresoras.

Durante los años 70 se utilizaban chips gráficos especializados en las máquinas recreativas de videojuegos, la razón de esto era que la *RAM* usada como *frame buffers* era demasiado cara, por lo que los chips gráficos escaneaban la salida por el monitor. Esto se puede ver en ejemplos como el **Fujitsu MB14241** el cual usaba un controlador para acelerar el dibujado de los *sprites* en juegos como *Gun Fight*(1975) ó *Space Invaders*(1978).

En la década de los años 80, ***Texas Instruments*** desarrolló **TMS34010**, el primer microprocesador con capacidad para gráficos en el mismo *chip*. Podía correr código de propósito general además de tener un conjunto de instrucciones muy orientadas a gráficos. Se empezaron a lanzar tarjetas conocidas como **VGA(Video Graphics Array)**, con una re-

solución de 640x480 y 256 colores distintos. El comercio de estas nuevas tarjetas supuso el comienzo de la *era gráfica*.

La popularidad que rápidamente cogieron los videojuegos allá por los años 90 y las exigencias del público y del mercado propició la aparición de las primeras tarjetas para el diseño de gráficos 3D, como las presentes en las videoconsolas de quinta generación *PlayStation* y *Nintendo 64*. Estas tarjetas seguían la evolución de **VGA**, el estándar **SVGA**, que implementaba funciones 3D. También en esa década llegaron muchas *APIs* para la programación de gráficos como **OpenGL** ó **DirectDraw** para aceleración por *hardware* de videojuegos 2D. A finales de esta década las tarjetas aceleradoras de gráficos 3D añadieron una etapa *hardware* al *pipeline* de renderización 3D siendo el primer ejemplo de esto la **Nvidia Geforce 256**.

En el cambio de siglo ***NVidia*** produjo el primer *chip* en el que se podían programar *shaders*. Aparte de esto, en Octubre de 2002, con la introducción en el mercado de la **ATI Radeon 9700** (también conocida como R300), la primera tarjeta con soporte **Direct3d 9.0**, empezaron a darse cuenta de que las *GPUs* llegarían a ser mucho más flexibles y rápidas para operaciones sobre imágenes que las *CPUs*. Se comenzaron a usar estas *GPUs* para otros fines que no sean los videojuegos y, dándose cuenta el colectivo científico del alto nivel de computo que tenían estas plataformas, se empezaron a usar para la aceleración de algoritmos y pruebas computacionales *High Performance*. Así, nació el paradigma que hoy conocemos como **GPGPU (General Purpose GPU)**.

Debido a los resultados tan buenos en rendimiento que estaban proporcionando (llegando casi a 100x de *SpeedUp*) se decidió mejorar lo que, por entonces, era el problema del uso de la *GPUs*: La dificultad de programar en ellas, ya que era necesarias *APIs* de programación de gráficos como **OpenGL** para explotar el paralelismo, lo cual limitaba el acceso a la

comunidad científica.

Para solucionar esto, ***NVidia*** invirtió mucho dinero para hacer que sus *GPUs* fueran totalmente programables y ofreció de forma totalmente transparente una *API* para la programación de sus *GPUs* en lenguajes como *C/C++*... Así, en 2006, ***NVidia*** presentó la nueva arquitectura, **CUDA** (*Compute Unified Device Architecture*), junto con una nueva arquitectura gráfica **G80** y la **GPU GeForce 8800 GTX**. Sin embargo, **CUDA** solo era útil para arquitecturas ***NVidia***, debido a esto surgió más tarde **OpenCL** de manos de ***Khronos Group***, para poder ser ejecutado en plataformas como *CPUs*, *GPUs* y *FPGAs*.

2.1.2. Estado del arte

Debido al enorme crecimiento de la industria del videojuego desde 2003 las gráficas han ido evolucionando hasta convertirse en potentes aceleradores. Los microprocesadores de las *GPUs* se han distanciado de las *CPUs* en lo que a cálculo en coma flotante se refiere.

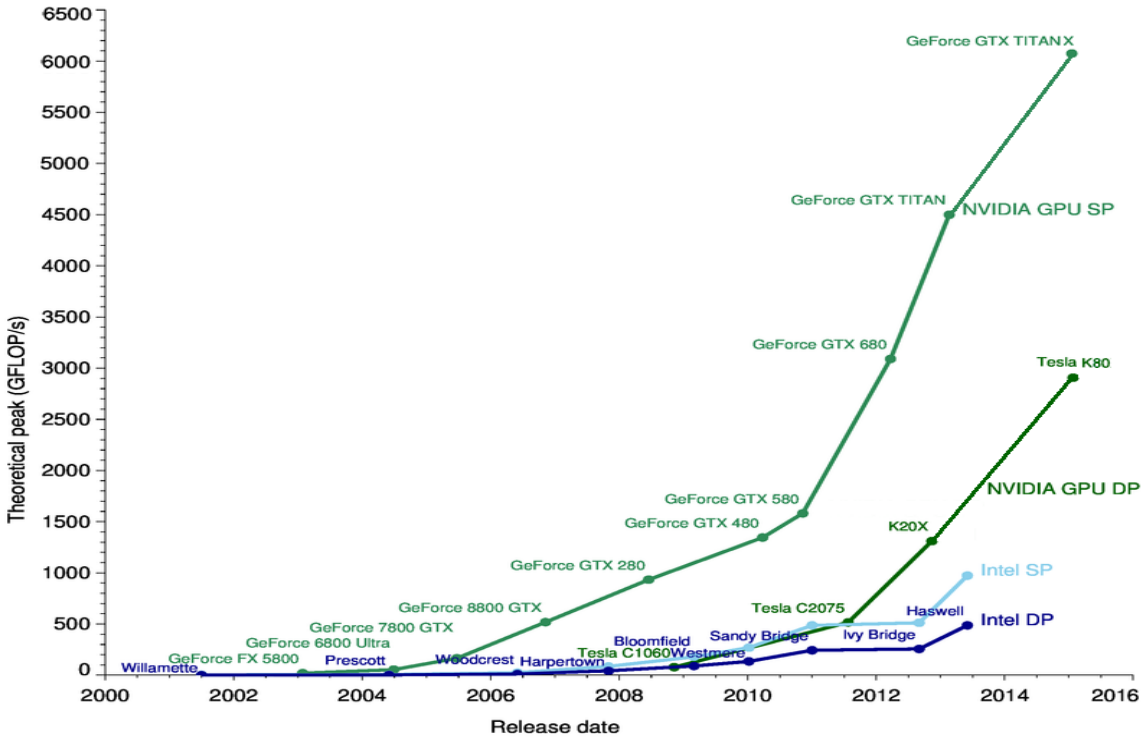

Figura 2.1: *Comparativa rendimiento en GFLOPS de procesadores y GPUs actuales.*

Como en la Figura 2.1 podemos observar, desde 2004 empezaba a haber un tímido distanciamiento en cuanto a *GFLOPS* entre las *GPUs* y las *CPUs* del momento. Todo esto se vio aumentado en 2006 cuando salió a la venta la **Geforce 8800 GTX** produciéndose un gran salto en el rendimiento de las *GPUs* con respecto a las *CPUs*. En la actualidad las GPUs proporcionan un rendimiento de 10 veces mayor que los procesadores *CPUs*, siendo capaz la gráfica más potente de procesar un poco más de 6 trillones de operaciones flotantes por segundo (unos 6 *TFLOPS* aproximadamente). Todo esto ha sido motivado debido a la especialización por parte de las *GPUs* en computación paralela e intensiva ya que están diseñadas de tal manera que se destinan la mayoría de los transistores a la lógica computacional, dotando a la misma de más unidades de computo en lugar de más jerarquías de memoria complejas como usan las *CPUs*. Al ser perfectas para resolver problemas con gran paralelismo de datos, en la literatura se consideran las *GPU* como procesadores vectoriales

modernos basados en el modelo **SIMD**(*Single Instruction Multiple Data*) aunque en realidad implementen un modelo **SIMT**(*Single Instruction Multiple Data*).

Tanto ***AMD/ATI*** como ***NVidia***, son dos compañías históricas en el campo del desarrollo de tarjetas gráficas, tanto de uso comercial como científico. Ambas apoyan y soportan *APIs* para que sus tarjetas se usen para cálculo paralelo.

En el año 2006 ***AMD*** presentó el repertorio de instrucciones **CMT**(*Close To Metal*) con el objetivo de competir con la arquitectura creada por ***NVidia***, su objetivo era el de aportar una forma de programación de propósito general en las tarjetas gráficas de la compañía. En 2008 después de ver que **CMT** tenía escaso apoyo, ***AMD/ATI*** optó por cancelar ese proyecto y apostar por **OpenCL**. En ese mismo año presentaron un marco de desarrollo que permitía trabajar sobre **OpenCL** como lenguaje de programación para poder utilizar las tarjetas gráficas de la compañía como procesadores de propósito general. ***NVidia*** desde 2006 ofrece algo similar con **CUDA**, siendo éste, además, un marco que permite el uso de **OpenCL**, un lenguaje específico para el procesado **GPGPU** sobre sus propias tarjetas.

Debido a su bajo coste y sus prestaciones, las tarjetas gráficas han experimentado un gran aumento de prestaciones. Podemos observar que no solo existen las comentadas anteriormente, sino que también existen modelos muy superiores orientados a la computación de altas prestaciones. Por ejemplo, existen *chips* como el **NVIDIA GK210** como el que usa la **NVIDIA K80**, que alcanza un rendimiento máximo de 8.74 *TFLOPS* con 4992 *cores* (2 x 2496 *cores*) y arquitectura **Kepler**, mientras que por ejemplo el procesador **Intel Core i7-3930** apenas alcanza 154 *GFLOPS*.

Como observamos en la Figura 2.2, el uso de este tipo de arquitecturas ha sufrido un crecimiento reseñable en la ámbito HPC (computación de altas prestaciones) quintuplicán-

dose el número de sistemas basados en este tipo de aceleradores entre los años 2011 y 2012.

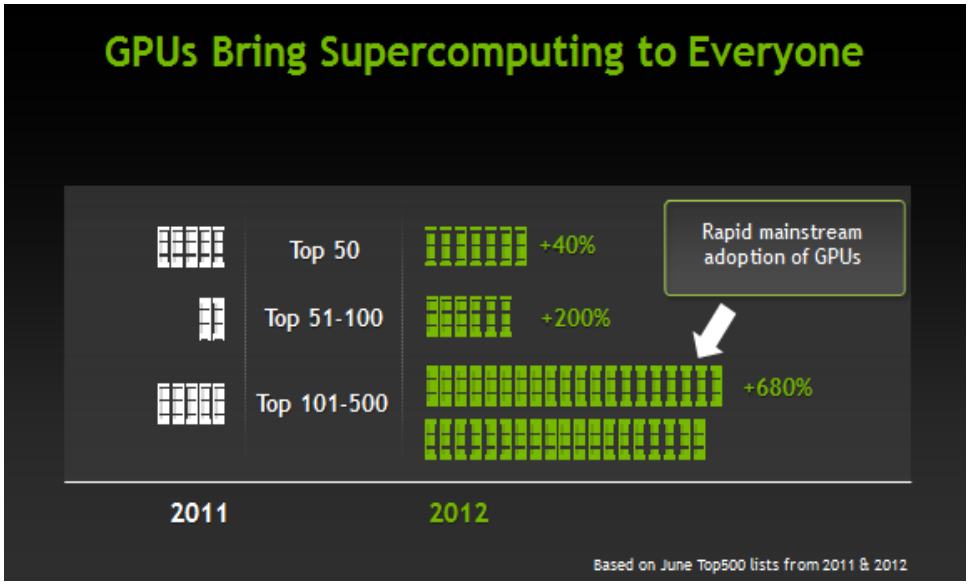

Figura 2.2: *Incremento de uso de GPUs en supercomputación.*

2.1.3. Otros aceleradores

Finalmente, mencionar la aparición de nuevas configuraciones de *CPU-GPU* usadas como aceleradores. En este apartado cabe destacar los nuevos procesadores de ***Intel(Xeon Phi)*** y los nuevos procesadores de ***AMD(A-series)***.

Intel sin participación en el mercado de los aceleradores presento los procesadores **Intel Xeon Phi** los cuales son tarjetas con formato **PCI** que funcionan sinérgicamente con los **Intel Xeon** para conseguir grandes ganancias de tiempo con código de masivo paralelismo el cual compensa la menor velocidad de funcionamiento.

La estrategia que ***Intel*** plantea en esto es el de sacar el máximo rendimiento de la mayoría de las aplicaciones con los procesadores **Intel Xeon**, reservando los Xeon Phi para todas aquellas aplicaciones que, como hemos dicho antes, tengan un alto grado de paralelismo. Al-

gunos ejemplos de este tipo de aplicaciones son: aplicaciones biológicas, meteorología...

Una de las mayores ventajas de estos componentes es que conservan un único código fuente entre los procesadores y los coprocesadores, solo es necesario optimizar una sola vez para aprovechar el paralelismo y así obtienen el máximo rendimiento en el procesador y en el coprocesador, por otro lado goza de mucha flexibilidad en la ejecución ya que a diferencia de una *GPU* normal el coprocesador puede alojar un sistema operativo, soportar estándares como *MPI* entre otras muchas cosas. La **Xeon Phi** soporta distintos modos de ejecución como son:

- **Modo simétrico**: lar cargas del trabajo se comparten entre el procesador anfitrión y el coprocesador.
- **Modo nativo**: la carga del trabajo reside completamente en el coprocesador y actúa fundamentalmente como un nodo independiente.
- **Modo descarga**: la carga del trabajo reside en el anfitrión y partes de esta se envían al coprocesador según se necesite.

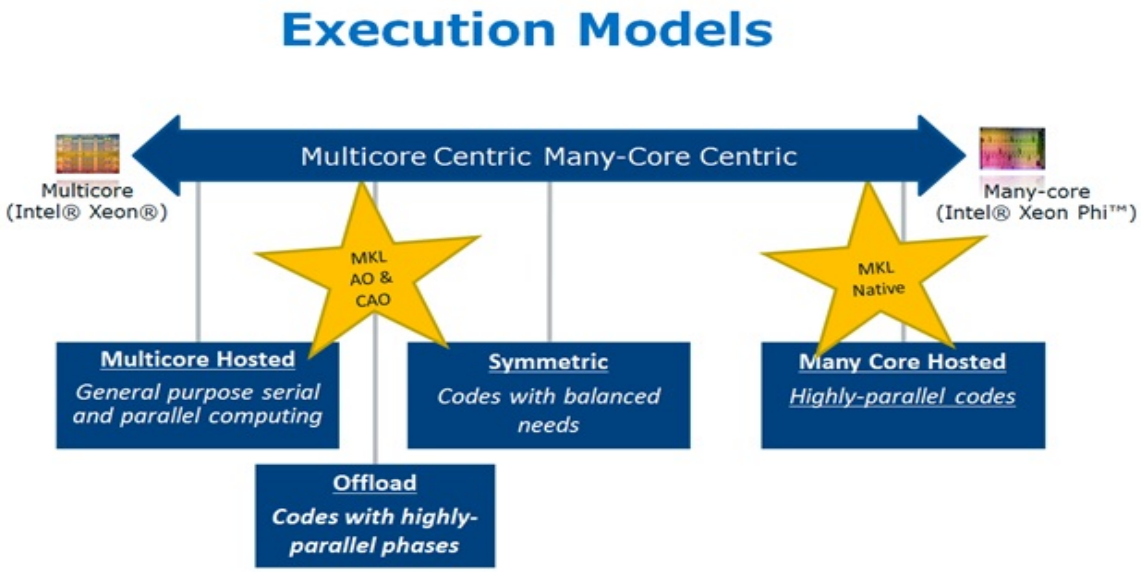

Figura 2.3: *Distintos modos de ejecución de* ***Xeon*** *y* ***Xeon Phi****.*

Por otro lado la empresa ***AMD/ATI*** esta centrada en el concepto de integración total de la *GPU* en el mismo circuito de la *CPU*, siendo ***AMD*** la única en usar este concepto de ***APU***(*Accelerated Processing Unit*) en lugar de usar el concepto de *CPU* clásico. Con esto intentan ofrecer una unidad de procesamiento que pueda trabajar con datos complejos de forma versátil.

La entrada de ***AMD*** en el mundo de las ***APUs*** fue propiciado por la adquisición de la empresa ***ATI*** por parte de ***AMD***, lo cuál hizo que ***AMD*** se distanciase de ***Intel*** en el mercado de las tarjetas gráficas integradas. Gracias a que este tipo de dispositivo están pensadas para trabajar con datos en paralelo, ciertas aplicaciones pueden aprovecharse de esta integración de *GPU* en la *CPU*, como pueden ser aplicaciones de generación de imágenes tridimensionales o el procesado de imágenes fotográficas.

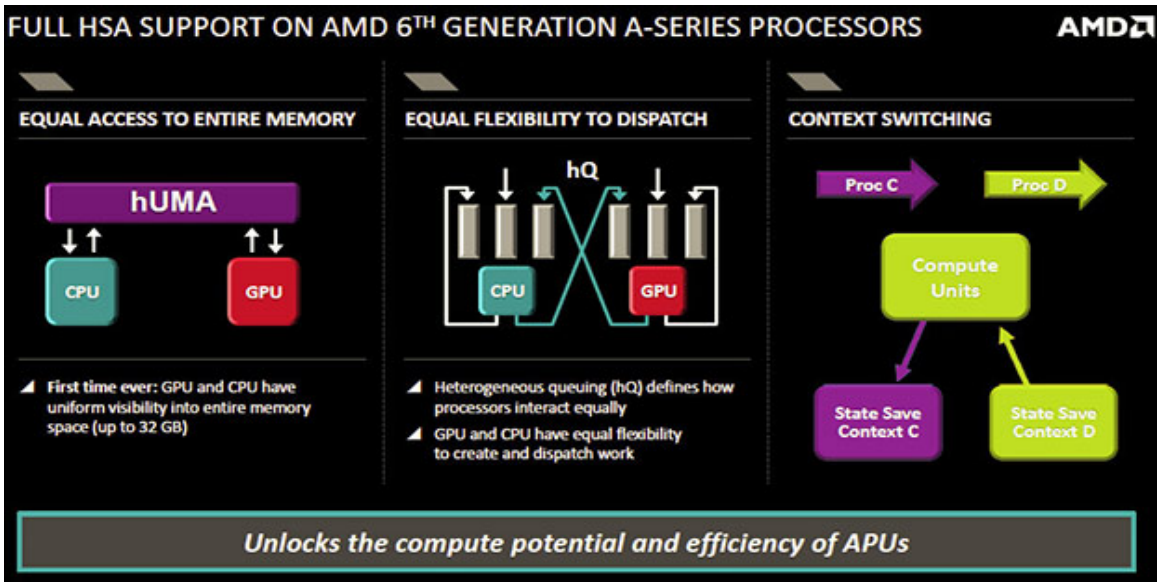

Figura 2.4: *Características de las APUs.*

En la actualidad, la última ***APU*** desarrollada por ***AMD*** es la denominada **Carrizo**, una nueva generación de *chips* para ordenadores de sobremesa, portátiles y móviles con cuatro núcleos de *CPU* y ocho núcleos de *GPU*. Las anteriores ***APUs*** permitían que todos

los núcleos se repartiesen la carga de trabajo de forma eficiente, eliminando de esta forma las barreras que han existido históricamente entre el *hardware* integrado de procesamiento gráfico y los núcleos de procesamiento central. En **Carrizo *AMD*** se ha volcado principalmente en la eficiencia energética aunque ha tenido otros objetivos como la mejora de predicción de saltos, mejora de caches... Aparte de todo esto, ***AMD*** está volcada en el concepto de núcleos de cálculo, procesadores cuyas capacidades de cálculo gráfico y de datos generales serán totalmente intercambiables desarrollando ambas funciones en igualdad de condiciones, más allá de su propia especialización. Un ejemplo de esto es poder acceder a memoria del sistema directamente para asumir tareas de procesamiento intensivo cuando la *CPU* integrada no puede hacerse cargo de ellas.

2.2. Métodos de Programación basados en directivas

El uso de estos métodos para la programación no es nada nuevo, existen desde hace tiempo multitud de compiladores que soportan estos sistemas de directivas, las cuales son utilizadas para guiar al compilador a la hora de generar código y facilitar la portabilidad de código. Aquí destaca **OpenMP**, el cuál genera código paralelo en sistemas con memoria compartida. **OpenMP** aparece en el año 2000 en los lenguajes *C/C++*. Su planteamiento fue la de unificar todas las soluciones de todos los fabricantes de sistemas de memoria compartida. Yéndonos al ámbito de la programación por directivas para *GPU* debemos destacar **OpenACC** [1], que apareció en Noviembre de 2011 y trata de repetir los pasos de los inicios de **OpenMP** pero en el campo de los aceleradores.

2.2.1. OpenMP

Es una *API* que soporta programación multiplataforma para sistemas de memoria compartida en *C, C++, Fortran*. Consiste en un conjunto de directivas de compilación, rutinas de bibliotecas y variables de entorno que influyen en la forma de programación en tiempo de compilación. **OpenMP** usa un modelo portable y escalable que mediante una interfaz

fácil y simple permite a los programadores desarrollar código paralelo para plataformas que van desde el escritorio de mesa hasta los supercomputadores, de hecho, puede usarse en conjunción con **MPI**[1] para *clusters* de computadores en sistemas de memoria compartida.

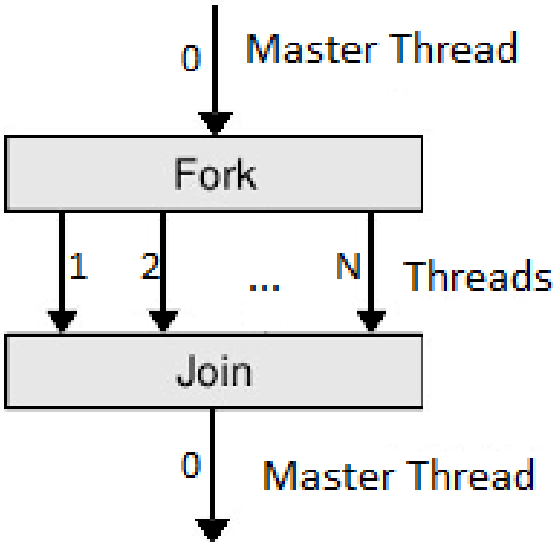

Figura 2.5: *Modelo de ejecución típico de OpenMP.*

En cuanto al modelo de ejecución de **OpenMP**, sigue el paradigma *fork-join* proveniente de los sistemas **UNIX**. Este paradigma se basa en dividir (*fork*) el problema en *N threads* (hilos), uniéndose (*join*) todos los resultados de los hilos finalizados en el hilo principal. Al basarse en directivas, cuando se introducen estas en un bloque de código este queda marcado como paralelo, incluyéndose una barrera al final de él para realizar la labor de sincronización de los distintos hilos, pudiéndose alterar este comportamiento mediante la directiva *nowait*.

[1] Es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.

```
int main(int argc, char **argv)
{
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

Figura 2.6: *Ejemplo de suma de vectores en CUDA.*

Como se puede ver en la Figura 2.6, el código multi-hilo para el ejemplo de una multiplicación de un vector/escalar es algo sumamente fácil de implementar. El uso de la directiva ***#pragma omp parallel for*** permite al compilador la paralelización del bucle *for* haciendo que cada iteración la ejecute un hilo independientemente del resto, con lo que un hilo hará $a[0] = 2 * 0$; otro $a[1] = 2 * 1$ ... y extendido a n: $a[n] = 2 * n$.

2.2.2. OpenACC

Hoy en día **OpenACC** [1] ha conseguido ser el principal modelo de programación de directivas para sistemas heterogéneos que incorporan aceleradores. Fue desarrollado por el grupo de compañías ***PGI, CAPS, NVIDIA y CRAY*** con el objetivo de simplificar la programación en sistemas heterogéneos *CPU/GPU*, además de facilitar la migración de código y el de ser una forma de programación muy barata.

Como se ha dicho anteriormente, **OpenACC** comparte premisas con **OpenMP**, proporcionando un conjunto de directivas que indica al compilador que zonas del código del *host* se deben paralelizar. Estas serán enviadas al *device* en forma de *kernels* tal y como se puede ver en la Figura 2.7. Al referirnos al *host* y al *device* nos referimos al *CPU* que ejecuta el código de forma secuencial y al acelerador que acelera las partes del código descritas por el programador.

Cuando se marca una región de código como paralelizable o se ponen datos que son objetivo de las directivas, ya sea para transferencia al dispositivo o por cualquier otro motivo, provocan que el compilador efectúe una labor de codificación y optimización, sacando el máximo rendimiento a las capacidades *hardware* del acelerador.

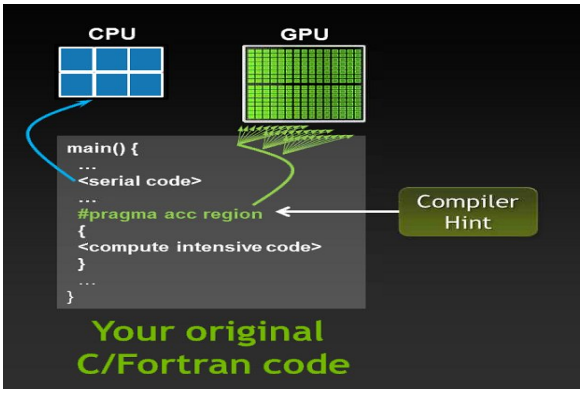

Figura 2.7: *Modelo de ejecución* ***OpenACC.***

Finalmente, nos gustaría destacar que el programador es responsable de conocer los diferentes tipos de niveles de paralelismo que ofrece su acelerador(**grano fino, grano grueso SIMD, operaciones vectoriales**). En el caso de bucles libres de dependencias se usará **grano grueso**, sin embargo en bucles con dependencias se usará **grano fino** o secuencial.

```
#pragma acc parallel
 for (i=0; i<N; i++)
 {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
 }
```

Figura 2.8: *Ejemplo de cálculo de Pi en* ***OpenACC.***

Como se puede ver en la Figura 2.8 el código para el cálculo de *pi* no requiere ape-

nas modificación del código fuente *C*, de forma análoga a como pasaba con **OpenMP**, la directiva ***#pragma acc parallel*** es el equivalente al visto en el ejemplo de **OpenMP**. Otra alternativa podría haber sido haber puesto la directiva ***#pragma acc kernels loop independent*** distribuye las iteraciones del bucle en un*kernel*.

2.3. Programación dedicada de GPUs

2.3.1. CUDA

Sobre el primer lustro de los 2000 surge el movimiento *GPGPU* motivado por el auge de las *GPUs* debido a su gran potencia de cálculo sobre las *CPUs* en operaciones con mucho paralelismo de datos. Este movimiento tenía unas limitaciones: gran esfuerzo de aprendizaje, sobrecarga *API* gráfico y repertorio limitado, limitaciones de acceso y emplazamiento en memoria y la existencia de muchos pasos de renderizado.

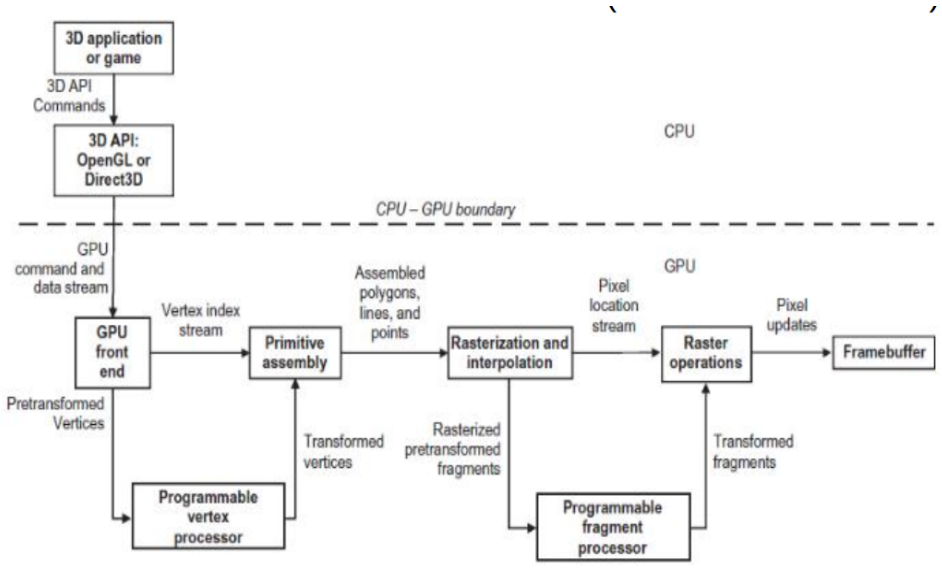

Figura 2.9: *Fases de ejecución.*

Bajo estas restricciones ***NVidia*** presentó en Noviembre del 2006 **CUDA** (*Compute Unified Device Architecture*) una arquitectura que permitía el uso de las *GPUs* de esa marca no solo para renderización de gráficos si no que también permitía hacer un uso más general de

esas *GPUs*, eliminando de un plumazo los problemas descritos anteriormente. Aún con esos problemas eliminados quedaban dos asuntos pendientes como la precisión en los resultados ya que no había *floats* (**FP32**).

Un código de **CUDA**, normalmente, se compone de un código *C/C++* con el que desarrollar el código del *host*, el código **CUDA** para ejecutar en la *GPU* destino y las librerías numéricas estándar como **FFT** o **BLAS**. Con esto el código de *C/C++* y **CUDA** se compila con el compilador NVCC de ***NVidia***, generando el pseudo-código ensamblador **PTX**, una vez se ha generado el **PTX**, el *driver* de la tarjeta gráfica lo traduce en el código ensamblador que será ejecutado en el *hardware* de destino.

kernel.cu

```
// Compute vector sum C = A+B in a kernel
__global__ void vecAddkernel(float* A_d, float* B_d, float* C_d,
    int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

Figura 2.10: *Ejemplo de suma de vectores en CUDA.*

Como se puede ver en la Figura 2.10, el código para una suma de vectores es algo sumamente fácil y es sumamente parecido a la sintaxis de *C*, ***__global__*** es un identificador que significa que este método puede ser llamado desde el *host* y desde el propio *device*. Los identificadores ***threadIdx.x***, ***blockDim.x*** y ***blockIdx.x*** son identificadores usados para obtener el indice del hilo dentro del bloque, el tamaño del bloque en la dimension x y el indice del bloque en la dimensión x. El resto de sintaxis es como en *C*.

2.4. Programación heterogénea. OpenCL

2.4.1. Historia

OpenCL (*Open Computing Language*) es un estándar abierto creado para programación paralela multiplataforma mediante un lenguaje de programación basado en **C99** de todo tipo de dispositivos: *CPU* (*Central Processing Unit*), *GPU* (Graphics Processing Unit), *DSP* (*Digital Signal Processors*), FPGA (Field-Programmable Gate Arrays) y otros procesadores o aceleradores.

Apple Inc creó la especificación original de **OpenCL** siendo desarrollada en conjunto junto con ***AMD, IBM, Intel y NVidia***. Cuando este la convirtió en un estándar abierto y libre de derechos la cedió al **Grupo Khronos**.El 16 de Junio de 2008 **Khronos** creó el **Compute Working Group** para llevar a cabo el proceso de estandarización.
OpenCL alcanzó mucha popularidad, tanto es así que ***AMD*** decidió abandonar su *API Close to Metal* por este motivo y por el escaso apoyo de los desarrolladores y redirigiendo sus esfuerzos hacia **OpenCL**. ***NVidia*** mediante la arquitectura **CUDA** soporta **OpenCL**, traduciendo en el fondo el código de **OpenCL** a código **CUDA** y sus sucesivas transformaciones a partir de este.

OpenCL evoluciono de forma rápida debido a lo anteriormente mencionado, extendiendo las compañías que la apoyan a ***Altera*** la cual traduce **OpenCL** directamente a puertas lógicas para que puedan ser ejecutado **OpenCL** directamente sobre sus **FPGAs** además de **ARM**, **Creative Technology**, **IBM**, **Qualcomm**...

2.4.2. Evolución

A continuación, un breve esquema sobre las diferentes versiones de **OpenCL** a través de su historia.

OpenCL 1.1

Fue lanzado el 14 de Junio del 2010 añadiendo una gran funcionalidad que mejoraba la flexibilidad, funcionalidad y el rendimiento de la programación paralela. Algunos de estos cambios fueron:

- Nuevos tipos de datos incluyendo los datos vector de 3 componentes y mas formatos para imágenes.
- Poder manejar comandos desde múltiples hilos del *host* y procesamiento de *buffers* a través de múltiples dispositivos.
- Operaciones en regiones de *buffers* incluyendo la lectura, escritura o copia de regiones 1D, 2D o áreas 3D rectangulares.
- Mejora del uso de eventos para manejar el control de ejecución de comandos.
- Muchas más funciones como funciones de acotación, barajado de valores entre muchos otros.
- Mejora de la interoperabilidad con **OpenGL** a través de una forma de compartir más eficiente imágenes y *buffers* usando los eventos para comunicarlos.

OpenCL 1.2

Khronos Group anunció en Noviembre del 2015 la especificación de **OpenCL 1.2**, la cuál añadía una muchísima funcionalidad sobre las anteriores versiones en términos de rendimiento. Muchas de estas notables características son:

- El poder particionar un dispositivos en mini-dispositivos de tal forma que una tarea pueda ser alojada en unidades de computo individuales. Esto es útil para reservar áreas del dispositivo a fin de reducir la latencia de tareas de tiempo real.
- Separación de la compilación y el enlazado de objetos, pudiendo compilar **OpenCL** en librerias externas para poder incluirlos en otros programas.
- Mejora del soporte para imágenes
- *Built-in Kernels*.
- Soporte con *DirectX*.
- Adopción de IEEE 754.

OpenCL 2.0

En Noviembre del 2013 Khronos Group anunció el lanzamiento de OpenCL 2.0 añadiendo actualizaciones como son:

- Memoria compartida virtual.
- Paralelismo anidado.
- Espacio de direcciones genérico.
- Operaciones atómicas de C11.
- *Driver* para

OpenCL 2.1

En la **GDC (Game Developer Conference)** de San Francisco en Marzo de 2015 se anunció **OpenCL** 2.1 y se sustituyó **OpenCL** C por **OpenCL** C++ que tenía un subconjunto de C++14. Esta versión presentaba algunas características como:

- Funcionalidad de subgrupos.

- Copia de objetos *kernel* y estados.
- Baja latencia en las peticiones al dispositivo.
- Ingestión de código **SPIR-V** en ejecución.
- Ejecución por prioridades en las colas.
- Comparte con **Vulkan SPIR-V** como representación intermedia.

OpenCL 2.2

Mejora la productividad en OpenCL C++ con mejoras como:

- **SPIR-V 1.1** con pleno soporte para **OpenCL** C++.
- Mejoras para generar código más optimizado.
- Puede correr sobre cualquier *hardware* con soporte para **OpenCL 2.0** (Solo se necesita actualizar los *drivers*).

2.4.3. Modelo de ejecución

OpenCL se basa en el modelo *host-device* lo cual significa que cálculos menores y la preparación de los cálculos a paralelizar se harán en el *hardware* anfitrión y los cálculos se realizarán en los dispositivos.

Los dispositivos están compuestos por unidades de cómputo (CU) las cuales a su vez están formadas por elementos de procesamiento (PE). Por ejemplo, una *CPU* tendrá una unidad de cómputo por cada núcleo que tenga y cada unidad de cómputo tendrá 1 elemento de procesamiento o n si se están usando instrucciones **SIMD**.

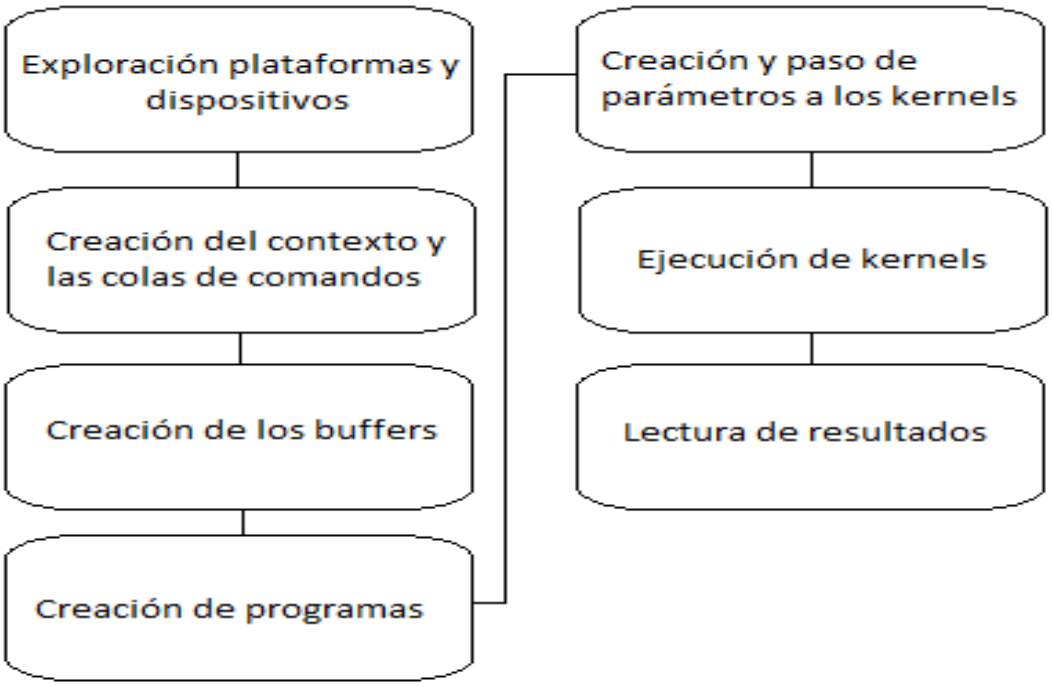

Figura 2.11: *Bloques básicos de código en* ***OpenCL****.*

Tal y como se puede observar en la Figura 2.11 **OpenCL** tiene unos bloques básicos de código, en los cuales se realizan las siguientes actividades:

- **Exploración de plataformas y dispositivos.** En este paso se identifica la lista de las plataformas que tienen *hardware* compatible con **openCL** presente en el equipo. Después de identificar las plataformas, para las plataformas deseadas por el usuario se identifican los dispositivos que hay de esa plataforma presentes en el equipo y se seleccionan los dispositivos en los que correr el código **OpenCL**.

- **Creación del contexto y las colas de comandos.** Se crean el contexto en el que estará enmarcada la aplicación y las correspondientes colas de comandos que ejecutaran las operaciones de ejecución de *kernels*, lectura de memoria...

- **Creación de *buffers*.** Se crean los *buffers* necesarios para albergar los datos de la

aplicación, por norma general se crea un *buffer* por cada *array* que se desea enviar al dispositivo.

- **Creación de programas.** Se crean los objetos de programas y se compilan en tiempo de ejecución los códigos **OpenCL**.
- **Creación de *kernels* y paso de parámetro a estos.** Se crean los objetos de *kernels*, los cuales serán los que se manden a ejecutar al dispositivo. Tras la creación de estos, se le deben de pasar los parámetros de la función del *kernel* a ejecutar.
- **Ejecución de *kernels*.** Se mandan los *kernels* creados en el paso anterior al dispositivo para que sean ejecutados por este.
- **Lectura de resultados.** Se leen los valores de salida de los *kernels* ejecutados en el paso anterior de vuelta al anfitrión para los motivos deseados por el usuario.

Los datos usados en los pasos anteriores son los siguientes:

- **cl_platform_id** es un tipo de dato que identifican las plataformas presentes en un equipo y **cl_device_id** es el tipo que identifica el dispositivo que se va a usar de una cierta plataforma.
- **cl_context** representa el entorno de **OpenCL** que engloba la gestión de la memoria y las colas de comandos cuyo tipo es definido por **cl_command_queue** que están asociados a un dispositivo y que almacenan un conjunto de operaciones a ser ejecutadas.
- El tipo **cl_mem** representa regiones de memoria en los dispositivos, se usan para guardar los datos que se pasan por parámetros a los *kernels* y hay disponibles operaciones sobre ellos como la operación de lectura y escritura.
- **cl_program**, es un tipo que representan un programa a ejecutar , se usan para compilar el código y construir los *kernels* que están representados con el tipo **cl_kernel**

que representa a los elementos a ejecutar en los dispositivos, se crean a partir de los objetos de programa y se necesita pasarles los parámetros.

2.4.4. Modelo de memoria

La organización de la memoria supone un reto para su descripción en **OpenCL** porque tiene que se válido para muchos tipos de dispositivos. El modelo de memoria hace distinción de 4 tipos tal y como se puede observar en la Figura 2.12:

- **Memoria global**. Es la memoria visible para todas las unidades de procesamiento del dispositivo, no posee consistencia entre los distintas unidades de procesamiento del dispositivo. Es un tipo de memoria con gran capacidad de almacenamiento pero con unos tiempos de acceso muy grandes.

- **Memoria de constantes**. Es una región específica de la memoria global destinada a los datos que son constantes. Presentan la mismas características que la memoria global pero con la diferencia que la cantidad de memoria destinada a este espacio es menor.

- **Memoria local**. Es visible solo dentro de una unidad de cómputo, es un tipo de memoria menos abundante en cuanto a capacidad que la memoria global pero mucho más rápida en cuanto al tiempo de lectura y escritura que esta.

- **Memoria privada**. Es visible solo para cada hilo de la unidad de cómputo, presenta las mismas características que la memoria local con la diferencia que está tiene mucho menos capacidad que la anterior y sus tiempos de acceso son bastante más bajos.

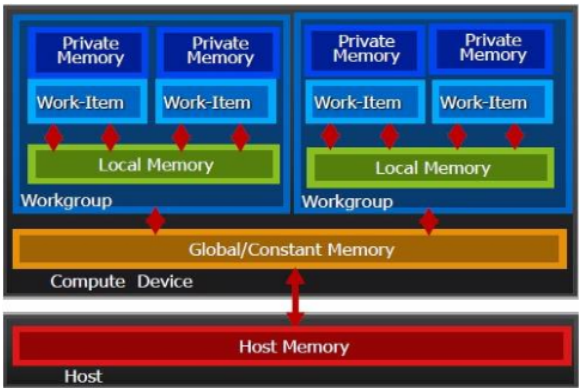

Figura 2.12: *Modelo de memoria de OpenCL.*

2.4.5. Ejemplo. Suma de vectores

A continuación vamos a presentar un ejemplo para ilustrar el funcionamiento de OpenCL. Vamos a ilustrarlo con el típico código de la suma de vectores[10].

En la primera figura de código encontramos las declaraciones e inicializaciones de las variables que necesita usar para el cálculo de la suma en **OpenCL**, como por ejemplo el código de la suma de vectores en **OpenCL** el cual introduce en el código de C en forma de *array de char* o la inclusión de la librería **cl.h**. El código del *kernel* se especifica en una cadena de caracteres cuya funcionalidad es análoga a un rutina convencional. Como se puede ver la sintaxis de **OpenCL** es tremendamente parecida a C, el código básicamente obtiene el id del hilo que lo esta ejecutando y realiza la suma de de las componentes de la posición *id*.

Algorithm 1 Ejemplo de suma de vectores en **OpenCL** - Inicializaciones.

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <CL/cl.h>
4: const char* programSource =
5: "__kernel \n"
6: "void vecadd(__global int *A, \n"
7: " __global int *B, \n"
8: " __global int *C) \n"
9: "{ \n"
10: " \n"
11: " int idx = get_global_id(0); \n"
12: " \n"
13: " C[idx] = A[idx] + B[idx]; \n"
14: "} \n"
15: ;
16: int main() {
17: int *A = NULL;
18: int *B = NULL;
19: int *C = NULL;
20: const int elements = 2048;
21: size_t datasize = sizeof(int)*elements;
22: A = (int*)malloc(datasize);
23: B = (int*)malloc(datasize);
24: C = (int*)malloc(datasize);
25: for int i = 0; i <elements; i++ do
26:    A[i] = i;
27:    B[i] = i;
28: end for
29: cl_int status;
```

Algorithm 2 Ejemplo de suma de vectores en **OpenCL** - Exploración de plataformas y dispositivos.

```
1: // STEP 1: Discover and initialize the platforms
2: cl_uint numPlatforms = 0;
3: cl_platform_id *platforms = NULL;
4: status = clGetPlatformIDs(0, NULL, &numPlatforms);
5: platforms = (cl_platform_id*)malloc(numPlatforms*sizeof(cl_platform_id));
6: status = clGetPlatformIDs(numPlatforms, platforms,NULL);
7: // STEP 2: Discover and initialize the devices
8: cl_uint numDevices = 0;
9: cl_device_id *devices = NULL;
10: status = clGetDeviceIDs(platforms[0],CL_DEVICE_TYPE_ALL,0,NULL,&numDevices);

11: devices =(cl_device_id*)malloc(numDevices*sizeof(cl_device_id));
12: status = clGetDeviceIDs(platforms[0],CL_DEVICE_TYPE_ALL,numDevices,devices,NULL);
```

En el siguiente fragmento de código estamos explorando las diferentes plataformas presentes en el equipo en el que se ejecuta el programa mediante la función **clGetPlatformIDs**, obteniendo en primer lugar el número de plataformas presentes en el equipo con el que poder dimensionar el array de plataformas **platforms** y en una segunda vez los indices de todas las plataformas.

En la segunda parte del código lo que estamos haciendo con el uso de la función **clGetDeviceIDs** es como antes en primer lugar obtener el número de dispositivos presentes de una cierta plataforma de nuestro equipo con el que dimensionar el array de dispositivos **devices** y en un segundo lugar inicializar ese array con el índice de todos los dispositivos de la primera plataforma que **OpenCL** haya encontrado en nuestro equipo, el término **CL_DEVICE_TYPE_ALL** indica a **OpenCL** que coja cualquier dispositivo de esa plataforma, otros valores serían CL_DEVICE_TYPE_CPU para indicarle que solo escoja *CPUs* **CL_DEVICE_TYPE_GPU** *GPUs...*

Algorithm 3 Ejemplo de suma de vectores en **OpenCL** - Creación del contexto y las colas de comandos.

```
1: // STEP 3: Create a context
2: cl_context context = NULL;
3: context = clCreateContext(NULL,numDevices,devices,NULL,NULL,&status);
4: // STEP 4: Create a command queue
5: cl_command_queue cmdQueue;
6: cmdQueue = clCreateCommandQueue(context,devices[0],0,&status);
```

En esta parte nos encontramos con la creación del contexto en primera instancia y de la cola de comandos en segunda estancia. Al crear el contexto mediante la función **clCreateContext** se le especifica los dispositivos que pertenecerán a ese contexto, en nuestro caso escogemos todos los que tomamos en la parte anterior que es lo que en el código aparece como *numDevices,devices* y como **NULL** estamos especificando las propiedades del contexto como pueden ser que solo admita dispositivos de una cierta plataforma. Los otros corresponden a funciones a ejecutar por **OpenCL** cuando se produzcan errores en la ejecución de esta función. Finalmente *status* se pasa como variable de entrada-salida para recoger el código de error devuelto por la función (0 éxito resto error). Para la creación de la cola de comandos le pasamos a **clCreateCommandQueue** el contexto al que esta asociada, el dispositivo al que se debe vincular esta cola de comandos y la variable *status* con la misma intención que con el contexto. El 0 pasado como parámetro es para indicar que queremos asignarle las propiedades por defecto a esta cola de comandos.

Algorithm 4 Ejemplo de suma de vectores en **OpenCL** - Creación de los buffers.

```
1: // STEP 5: Create device buffers
2: cl_mem bufferA; // Input array on the device
3: cl_mem bufferB; // Input array on the device
4: cl_mem bufferC; // Output array on the device
5: bufferA = clCreateBuffer(context,CL_MEM_READ_ONLY,datasize,NULL,&status);

6: bufferB = clCreateBuffer(context,CL_MEM_READ_ONLY,datasize,NULL,&status);

7: bufferC = clCreateBuffer(context,CL_MEM_WRITE_ONLY,datasize,NULL,&status);

8: // STEP 6: Write host data to device buffers
9: status = clEnqueueWriteBuffer(cmdQueue,bufferA,CL_FALSE,0,datasize,A,0,NULL,NULL);

10: status = clEnqueueWriteBuffer(cmdQueue,bufferB,CL_FALSE,0,datasize,B,0,NULL,NULL);
```

En este fragmento presenciamos la creación de los *buffer* del dispositivo, tal como dijimos antes, lo normal (sin optimizaciones) es crear un *buffer* por cada array del *kernel* con lo que como tenemos tres parámetros que corresponde con los 3 *buffers*. La forma de crear un *buffer* en **OpenCL** es con la función **clCreateBuffer** especificándole el contexto en el que van a ser creados, las *flags* del *buffer* las cuales indican el modo de acceso al *buffer* si va a ser inicializado al ser creado, el tamaño del *buffer*, los datos si queremos inicializarlo en el mismo momento en el que creamos el *buffer*, en nuestro caso para mostrar como es la escritura elegimos no hacerlo pasándole **NULL** (en caso de querer hacerlo hay que pasarle **CL_MEM_WRITE_ONLY|CL_MEM_COPY_HOST_PTR** en lugar de **CL_MEM_WRITE_ONLY**) y por último la variable para comprobar el código de salida. En el caso de la escritura en los *buffers* se realiza con la función **clEnqueueWriteBuffer** en la que especificamos la cola de comandos en la que va a ser encolada la operación, el *buffer* en el que queremos escribir, si queremos que la llamada sea síncrona o asíncrona (en nuestro caso elegimos asíncrona), si queremos elegir escribir con un cierto offset inicial, los datos del anfitrión a escribir, y los tres últimos parámetros tienen que ver con la sincronización de tareas mediante eventos.

Algorithm 5 Ejemplo de suma de vectores en **OpenCL** - Creación de los programas.

```
1: // STEP 7: Create and compile the program
2: cl_program program = clCreateProgramWithSource(context,1,(const
   char**)&programSource,NULL,&status);
3: status = clBuildProgram(program,numDevices,devices,NULL,NULL,NULL);
```

En esta parte del código estamos compilando el programa de la suma de vectores que definimos anteriormente como un *array de char*. Mediante **clCreateProgramWithSource** creamos el programa especificándole el contexto al que va a estar asociado, el numero de punteros a *array de char* que vamos a especificarle, el puntero a los *arrays* de char que contienen los programas, un *array* con las longitudes de cada *string* (en nuestro caso no lo especificamos) y por ultimo se le puede pasar una variable para que te devuelva el código de error.

Finalmente con **clBuildProgram** lo que hacemos es compilar el programa especificando el programa a compilar, el número de dispositivos en los que compilarlo, los dispositivos en los que compilarlo, las opciones de compilación, y como sucedía al crear contexto los dos últimos parámetros son para asignarle un puntero a función que ejecutará en el caso de producirse un error.

Algorithm 6 Ejemplo de suma de vectores en **OpenCL** - Creación y paso de parámetros a los kernels.

```
1: // STEP 8: Create the kernel
2: cl_kernel kernel = NULL;
3: kernel = clCreateKernel(program, "vecadd", &status);
4: // STEP 9: Set the kernel arguments
5: status = clSetKernelArg(kernel,0,sizeof(cl_mem),&bufferA);
6: status j= clSetKernelArg(kernel,1,sizeof(cl_mem),&bufferB);
7: status j= clSetKernelArg(kernel,2,sizeof(cl_mem),&bufferC);
```

Después de crear el programa y de haberlo compilado, es el turno de crear los *kernels* con la función **clCreateKernel** a la cual le especificamos el programa a partir del cual crear el *kernel* y el nombre del *kernel* a vincular al objeto. Después de crear el *kernel* se le deben de especificar los argumentos mediante **clSetKernelArg** al cual se le pasa el objeto del *kernel* al cual queremos especificarle los parámetros, el indice del parámetro a especificar, el tamaño del parámetro, y la dirección del parámetro a especificar.

Algorithm 7 Ejemplo de suma de vectores en OpenCL - Ejecución de los kernels.

```
1: // STEP 10: Configure the work-item structure
2: size_t globalWorkSize[1];
3: globalWorkSize[0] = elements;
4: // STEP 11: Enqueue the kernel for execution
5: status = clEnqueueNDRangeKernel(cmdQueue,kernel,1,NULL,globalWorkSize,NULL,0,NULL,NULL)
```

Una vez que ya tenemos todo los preparativos hechos es hora de mandar al *kernel* a ejecutar al dispositivo mediante la llamada a **clEnqueueNDRangeKernel** al cual se le pasa la cola de comandos en la que encolar esta operación, el *kernel* a ejecutar, el número de dimensiones del problema (como nuestra operación es sobre un array 1D pues metemos un 1), si queremos que los indices de los hilos empiecen con un cierto offset, el tamaño global de las dimensiones del problema, si queremos especificar un tamaño para el *WorkGroup*, y los tres últimos parámetros es para poder organizar las ejecuciones de operaciones mediante eventos.

Algorithm 8 Ejemplo de suma de vectores en OpenCL - Lectura de resultados y limpieza de datos.

```
1: // STEP 12: Read the output buffer back to the host
2: clEnqueueReadBuffer(cmdQueue,bufferC,CL_TRUE,0,datasize,C,0,NULL,NULL);
3: bool result = true;
4: for int i = 0; i <elements; i++ do
5:    if C[i] != i+i then
6:       result = false;
7:       break;
8:    end if
9: end for
10: if result then
11:    printf(.°utput is correct\n");
12: else
13:    printf(.°utput is incorrect\n");
14: end if
15: // STEP 13: Release OpenCL resources
16: // Free OpenCL resources
17: clReleaseKernel(kernel);
18: clReleaseProgram(program);
19: clReleaseCommandQueue(cmdQueue);
20: clReleaseMemObject(bufferA);
21: clReleaseMemObject(bufferB);
22: clReleaseMemObject(bufferC);
23: clReleaseContext(context);
24: // Free host resources
25: free(A);
26: free(B);
27: free(C);
28: free(platforms);
29: free(devices); }
```

Finalmente, mediante una llamada igual a la de escritura de un *buffer* hacemos una lectura del *buffer* donde hemos guardado el resultado de ejecutar el *kernel* en el dispositivo y comprobamos si ha salido el resultado esperado.

Capítulo 3

Algoritmos, métricas y estímulos

3.1. Nuestros algoritmos

A continuación, vamos a introducir los algoritmos que hemos implementado para la realización del trabajo. Se han escogido dos algoritmos del modelo de gradiente por ser estos algunos de los más utilizados y que además son actualmente empleados en disciplinas diversas como la Inteligencia Artificial o la biología.

3.2. Lucas&Kanade

El algoritmo de **Lucas&Kanade** es todo un clásico en los modelos de gradiente. Debido al límite de nuestros recursos computacionales, resulta el algoritmo perfecto, pues solo es necesario el almacenamiento de un filtro espacial (que no destacará por su tamaño) que irá recorriendo la imagen de *pixeles*, reduciendo el problema a la convolución de una imagen.

Lucas&Kanade presupone que el desplazamiento en un flujo ocurre entre dos instantes cercanos, por consiguiente, son los *pixeles* vecinos los que poseerán la información del desplazamiento. Para tratar esta vecindad, se hace uso de una serie de filtros que se usarán para calcular las derivadas del movimiento, ya sea espacial o temporalmente.

Así pues, la velocidad: (V_x, V_y) de un *pixel* debe satisfacer:

$$\begin{aligned} I_x(q_1)V_x + I_y(q_1)V_y &= -I_t(q_1) \\ I_x(q_2)V_x + I_y(q_2)V_y &= -I_t(q_2) \\ &\vdots \\ I_x(q_n)V_x + I_y(q_n)V_y &= -I_t(q_n) \end{aligned} \tag{3.1}$$

Donde $q_1, q_2, ..., q_n$ son los *pixeles* y $I_x(q_i), I_y(q_i), I_t(q_i)$ son las derivadas parciales de la imagen I con respecto al eje x, al eje y (derivadas espaciales) y al tiempo t. Esta ecuación solo permite estimar la velocidad en la dirección del gradiente máximo. Para resolver este problema, se construye una estimación basada en las derivadas de primer orden por medio de los *Mínimos Cuadrados*. El modelo extrae la estimación bajo la hipótesis de que las velocidades en la vecindad de un *pixel* central son similares.

$$min \sum_{x \epsilon \Omega} W^2(x) = [I(x,y,t).(V_x, V_y) + I_t(x,y,t)]^2 \tag{3.2}$$

Donde $W(x)$ son los pesos que serán asignados a los *pixeles* de la vecindad Ω. Es mejor darle prioridad a los *pixeles* que están cerca, así, aquellos *pixeles* limítrofes tendrá mayor peso que aquellos más alejados o incluso en una posición $(x+1, y+1)$.

La solución al problema viene dada por:

$$\vec{V} = \left[A^T W^2 A\right]^{-1} A^T W^2 \vec{b} \tag{3.3}$$

Lo que al desarrollarlo resulta en un sistema de matrices que sencilla solución mediante simples multiplicaciones.

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \sum_i I_x(q_i)^2 & \sum_i I_x(q_i) I_y(q_i) \\ \sum_i I_y(q_i) I_x(q_i) & \sum_i I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i I_x(q_i) I_t(q_i) \\ -\sum_i I_y(q_i) I_t(q_i) \end{bmatrix} \tag{3.4}$$

Una de las limitaciones de este modelo se produce en el llamado problema de apertura. En estos casos, la matriz de la expresión 3.3 no se puede invertir debido a una división entre 0 que se intenta realizar.

Para concluir, la estimación entre dos *frames* se resuelve de acuerdo a la ecuación 3.4, siendo $\vec{v} = (v_x, v_y)$ la velocidad, que será calculada mediante la multiplicación de matrices.

3.3. Horn&Schunck

3.3.1. Modelo

El algoritmo de Horn&Schunk es un algoritmo de flujo óptico basado en dos presunciones.

La primera presunción es la llamada presunción de la constancia del brillo, la cual asume que todos los objetos de la escena tendrán un brillo el cual permanecerá constante.

Siendo ω una imagen y $I(x, y, t)$ el brillo de un *pixel* (x, y) en el instante t, el cual puede ser considerado como el número del *frame*. Siendo $(u(x, y), v(x, y))$ el campo desplazado definido en ω, entonces la presunción de la constancia del brillo se define como:

$$I(x + u, y + v, t + 1) = I(x, y, t) \tag{3.5}$$

La segunda presunción es que los *pixeles* vecinos se mueven de forma similar. En otras palabras, el flujo del campo es suave. Una de las formas de describir grado de suavidad que tiene en un punto particular es computar el cuadrado de la velocidad del gradiente.

Las derivaciones puede ser penalizadas con la siguiente funcional de energía:

$$\begin{array}{c} E_b(u,v) = \int_\omega (I(x+u, y+v, t+1) - I(x,y,t))^2 dxdy \\ E_s(u,v) = \int_\omega ||\delta u||^2 + ||\delta v||^2 dxdy \end{array} \tag{3.6}$$

La suma ponderada de esas funciones nos da la idea de como de bueno es un particular campo desplazado. El problema está en encontrar un punto mínimo global de la siguiente funcional de energía.

$$E(u,v) = E_b(u,v) + \alpha E_s(u,v), \alpha > 0 \tag{3.7}$$

El *Horn&Schunk* original se usa una version lineal de 3.5:

$$\begin{array}{c} I(x+u, y+v, t+1) \approx I(x,y,t) + I_x(x,y,t) \cdot u + I_y(x,y,t) \cdot v + I_t(x,v,t) \\ I(x+u, y+v, t+1) - I(x,y,t) \approx I_x u + I_y v + I_t \end{array} \tag{3.8}$$

Y del 3.6 obtenemos

$$E_b(u,v) = \int_\omega (I_x u + I_y v + I_t)^2 dxdy \tag{3.9}$$

De acuerdo al cálculo de las variaciones, u y v deben satisfacer las ecuaciones de *Euler-Lagrange* con las condiciones de reflejos de bordes *(Neumann)*. El campo desplazado puede ser solución de la fórmula de abajo.

$$\begin{array}{c} (I_x u + I_y v + I_t) I_x - \alpha(u_{xx} + u_{yy}) = 0 \\ (I_x u + I_y v + I_t) I_y - \alpha(v_{xx} + v_{yy}) = 0 \end{array} \tag{3.10}$$

Aclaración 1: $\delta u = u_{xx} + u_{yy}$

El problema con este enfoque es que se asumen pequeños desplazamientos de *píxeles* (debido a la primera derivada de los polinomios de *Taylor*). Hasta el momento tenemos la

estimación (u, v) hasta un pequeño incremento desconocido (du, dv). Usando el enfoque de *Horn&Schunk* convencional podemos encontrar un (du, dv) como un campo desplazado entre $I(x, y, t)$ y $I(x + u, y + v, t + 1)$:

$$\begin{array}{c} \tilde{I}(x, y, t + 1) =^{def} I(x + u, y + v, t + 1) \\ I(x + du, y + dv, t + 1) \approx \tilde{I}_x(x, y, t + 1)du + \tilde{I}_y(x, y, t + 1)dv + \tilde{I}(x, y, t + 1) \\ E_b(du, dv) = \int_\omega (\tilde{I}_x(x, t + 1)du + \tilde{I}_y(x, t + 1)dv + \tilde{I}(x, t + 1) - I(x, t))^2 dx \end{array} \tag{3.11}$$

Finalmente tenemos una ecuación diferencial para du y dv.

$$\begin{array}{c} (\tilde{I}_x du + \tilde{I}_y dv + \tilde{I} - I)\tilde{I}_x - \alpha(du_{xx} + du_{yy}) = 0 \\ (\tilde{I}_x du + \tilde{I}_y dv + \tilde{I} - I)\tilde{I}_y - \alpha(dv_{xx} + dv_{yy}) = 0 \end{array} \tag{3.12}$$

Aclaración 2: $\gamma(x, y, t + 1)$ es $I(x, y, t + 1)$ deformado con el vector de campo (u, v)

Una vez que tenemos el incremento desconocido, u y v pueden ser actualizadas. Podemos estar refinando el campo desplazado hasta que converja. Ese enfoque incremental puede combinarse con la estrategia de grano fino o estrategia multinivel. Empezamos resolviendo el problema para imágenes de dimensiones reducidas, donde los desplazamientos más largos serán del orden del *pixel*, y usamos la solución de pasar a imágenes de resoluciones más grandes. De esta forma podemos resolver el problema original paso a paso.

Poniendo todas las ideas juntas se sigue los siguientes pasos:

1. Preparar las versiones reducidas de las imágenes originales.
2. Inicializar la u y la v a cero.
3. Seleccionar la mínima resolución.
4. Deformar la imagen $I(x, y, t + 1)$ con la u y v actual.
5. Computar la $\tilde{I}_x$, $\tilde{I}_y$ y $\tilde{I}$ - I.

6. Resolver 3.9 para du, dv.

7. Actualizar u con du y v con dv.

8. Si du y dv no son los suficientemente pequeños, ir al paso 4.

9. Si la resolución actual es menor que la original, procedemos a aumentamos la resolución y volvemos al paso 4.

Todos los pasos del 4 y 7 (inclusive) se le llama iteraciones de deformación.

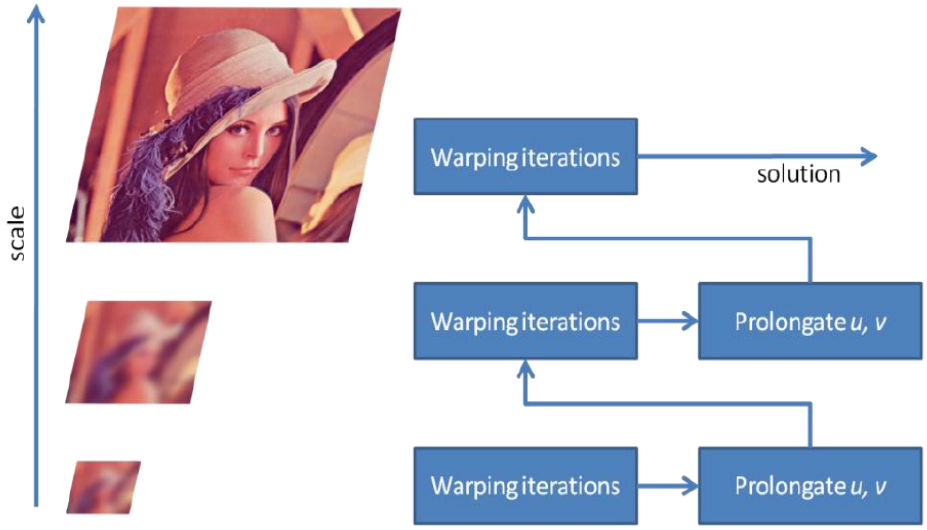

Figura 3.1: *Esquema del método.*

3.3.2. Resolviendo PDE

El problema más complejo del algoritmo es resolver el sistema de las ecuaciones diferenciales parciales de du y dv. Para ello se aplica el método de diferencias finitas.
El operador de *Laplace* está aproximado con la plantilla estándar de 5 puntos.

$$(\delta u)_{i,j} \approx (u_{-i,j} - 2u_{i,j} + u_{i+1,j}) + (u_{i,j-1} - 2u_{i,j} + u_{i,j+1}) \tag{3.13}$$

Aclaración: En procesamiento de imágenes el espaciado de la cuadrícula es normalmente

igual a 1

Usando la ecuación anterior podemos reescribir las ecuaciones diferenciales parciales de du y dv como un sistema de ecuaciones lineales

$$\begin{aligned}(\tilde{I}_x du_{ij} + \tilde{I}_y dv_{ij} + \tilde{I} - I)\tilde{I}_x - \alpha(\Delta du)_{ij} = 0 \\ (\tilde{I}_x du_{ij} + \tilde{I}_y dv_{ij} + \tilde{I} - I)\tilde{I}_y - \alpha(\Delta dv)_{ij} = 0\end{aligned} \tag{3.14}$$

Estas ecuaciones pueden ser solucionadas aplicando el método *Jacobi* [11]

$$\begin{aligned}du_{ij}^{n+1} = \bar{du}_{ij}^{n} - \frac{\tilde{I}_{x,ij}(\tilde{I}_{x,ij}\bar{du}_{ij}^{n} + \tilde{I}_{y,ij}\bar{dv}_{ij}^{n} + \tilde{I}_{ij} - I_{ij})}{\alpha + I_{x,ij}^2 + I_{y,ij}^2} \\ dv_{ij}^{n+1} = \bar{dv}_{ij}^{n} - \frac{\tilde{I}_{y,ij}(\tilde{I}_{x,ij}\bar{du}_{ij}^{n} + \tilde{I}_{y,ij}\bar{dv}_{ij}^{n} + \tilde{I}_{ij} - I_{ij})}{\alpha + I_{x,ij}^2 + I_{y,ij}^2}\end{aligned} \tag{3.15}$$

Donde:

$$\begin{aligned}\bar{du}_{ij}^{n} =^{def} \tfrac{1}{4}(du_{i-1j}^{n} + du_{i+1j}^{n} + du_{ij-1}^{n} + du_{ij+1}^{n}) \\ \bar{dv}_{ij}^{n} =^{def} \tfrac{1}{4}(dv_{i-1j}^{n} + dv_{i+1j}^{n} + dv_{ij-1}^{n} + dv_{ij+1}^{n})\end{aligned} \tag{3.16}$$

Las condiciones de los bordes se aproximan de la siguiente forma:

$$du_{ij} = \begin{cases} du(i,j), 0 \leq i < W, 0 \leq j < H \\ du(0,j), i = -1, 0 \leq j < H \\ du(W-1,j), i = W, 0 \leq j < H \\ du(i,0), 0 \leq i < W, j = -1 \\ du(i,H-1), 0 \leq i < W, j = H \end{cases} \tag{3.17}$$

Lo mismo sucede para dv.

3.3.3. Calculando las derivadas de la imagen

Las derivadas de la imagen se aproximan mediante la derivada numérica o filtro derivado de 5 puntos $\frac{1}{12}(-1, 8, 0, -8, 1)$[19]. A modo de ejemplo, $\tilde{I}_x(i,j)$ es reemplazado por $\frac{1}{2}(\tilde{I}_x(i,j,t+1) + I_x(i,j,t))$. Para los puntos fuera del dominio de la imagen se procede a su reflexión en los bordes.

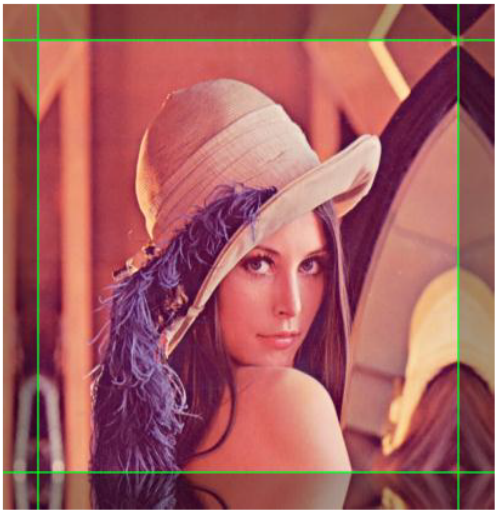

Figura 3.2: *Forma de manejar fuera de rango, los bordes están marcados con verde.*

3.3.4. Restricciones y prolongaciones

La restricción y la prolongación son dos de las operaciones más importantes en el enfoque multinivel. Estos términos surgieron de la teoría de los métodos multicuadrícula. La restricción es la responsable de, partiendo de unas cuantas rejillas, transformarlas en una de otra dimensión. La prolongación es el fenómeno inverso. La restricción se conoce como *downsampling*.

Tal y como se muestra en *Black&Sun* un factor de disminución de la imagen de 0.5 debería proporcionar una buena solución. Antes de aplicar *downsampling* a la imagen debemos aplicarle la técnica de *blur* para eliminar el *aliasing*. Haciendo la media en rejillas de $2x2$ casillas proporciona muy buena calidad.

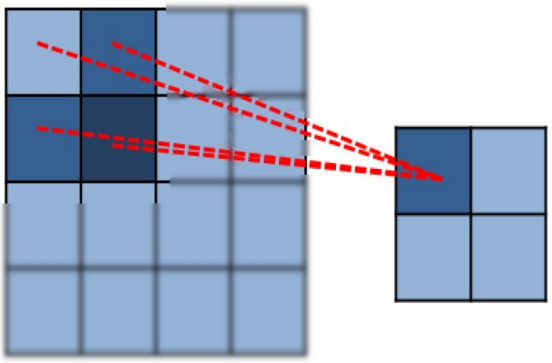

Figura 3.3: *Contracción por downsampling.*

La implementación de la prolongación corresponde a una interpolación bilineal seguida de un escalado apropiado. Ya que al hacer *downscale* usamos un factor de 0.5, necesitamos interpolar el vector por un factor de $\frac{1}{0,5} = 2$.

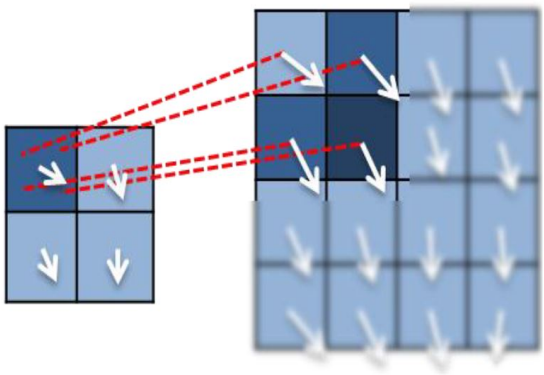

Figura 3.4: *Prolongación.*

3.4. Estimulos

Llamamos estímulos a los datos de entrada utilizados para comprobar que los algoritmos de flujo óptico y estimación de movimiento funcionan en cuanto a unas especificaciones das. Debido a esto, son bastante reconocidos los estímulos de carácter sintético al ser estos realizados artificialmente. Cuando se crea un estimulo, también se aportan los valores reales que debería ser capaz de captar un algoritmo. Por medio de métricas se puede calcular la

diferencia entre el error generado y el error real.

Por convenio, se suele representar el flujo como un campo de vectores. A este campo se le denomina *ground-truth*[16] y será usado junto con algunas métricas que se explicarán más adelante para comprobar la precisión de nuestros algoritmos.

A continuación pasaremos a explicar algunos de los estímulos que usaremos en el trabajo y que, por tanto, mencionaremos a lo largo de la memoria.

Primero debemos mencionar los estímulos de *DivergingTree* y *TranslatingTree*. Estos estímulos representan el zoom hacia un árbol y el movimiento horizontal de la cámara sobre un árbol respectivamente. Estos serán la base de estudio de nuestro trabajo, ya que han sido usados para todas las pruebas.

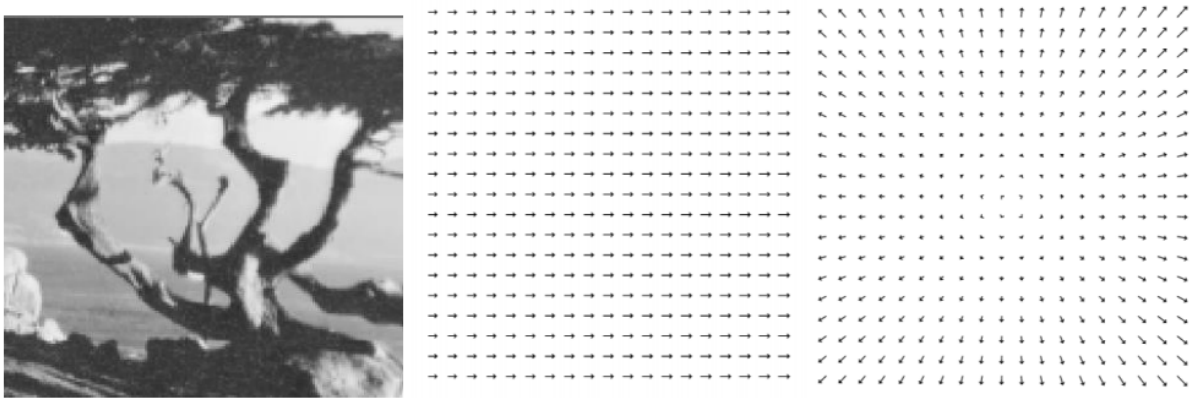

Figura 3.5: *Campo del flujo óptico para el caso de TranslatingTree y DivergingTree.*

También resulta imprescindible hablar del dataset de la página oficial de MiddleBury [5]. Desde esta plataforma se puede descargar un amplio conjunto de ejemplos de entrenamiento que, mediante un procesado previo, se pueden convertir en una película y en un *ground-truth* dividido en las componentes del vector.

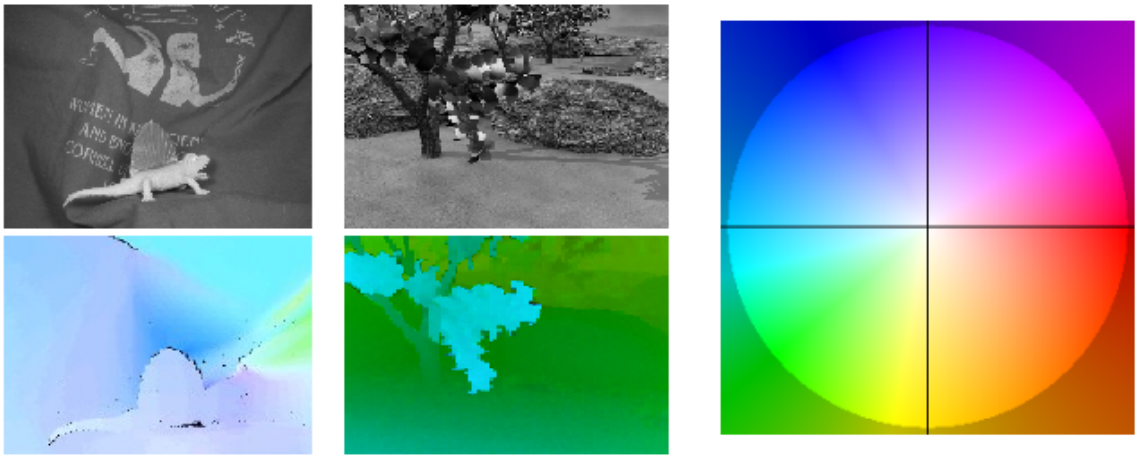

Figura 3.6: *Dos de los ejemplos de entrenamiento más sus ground-truth.*

En la Figura 3.6 podemos apreciar dos de los ejemplos de entrenamiento junto a sus *ground-truth* codificados cromáticamente por sus componentes vectoriales. La esfera que encontramos a la derecha indica los 4 cuadrantes de los que se forma cualquier plano.

También queremos destacar el *dataset* de **KITTI vision benchmark** [13] por tener un *dataset* de 2015 con el cual están haciendo pruebas para coches inteligentes y otros proyectos de Inteligencia Artificial.

3.5. Métricas

Las métricas nos ayudarán a especular cuán preciso es nuestro algoritmo comparando el resultado de nuestra implementación, al que llamaremos *resultado estimado*, con el resultado real del estímulo.

Para la correcta visualización del error, no se tomará una sola métrica, si no dos: la métrica de *John Barron* [5, 6] y la métrica de *Otte y Nagel* [5, 18]; midiendo el error angular y el error lineal respectivamente.

3.5.1. Error de Barron (Angular Error)

El error angular mide la diferencia en términos de amplitud del ángulo que forman el vector estimado con el real. La cantidad de movimiento de un *pixel* o velocidad se puede entender como el vector $v = (u, v)$ *pixels/frame* o como un vector dirección espacio-temporal de la forma (u, v, n), en unidades *(pixel, pixel, frame)*.

$$AE = cos^{-1}\left(\frac{1{,}0 + uxu_{GT} + vxv_{GT}}{\sqrt{1{,}0 + u^2 + v^2}\sqrt{1{,}0 + u_{GT}^2 + v_{GT}^2}}\right) \tag{3.18}$$

Mediante la Ecuación 3.18 podemos apreciar el cálculo necesario para resolver el error siendo

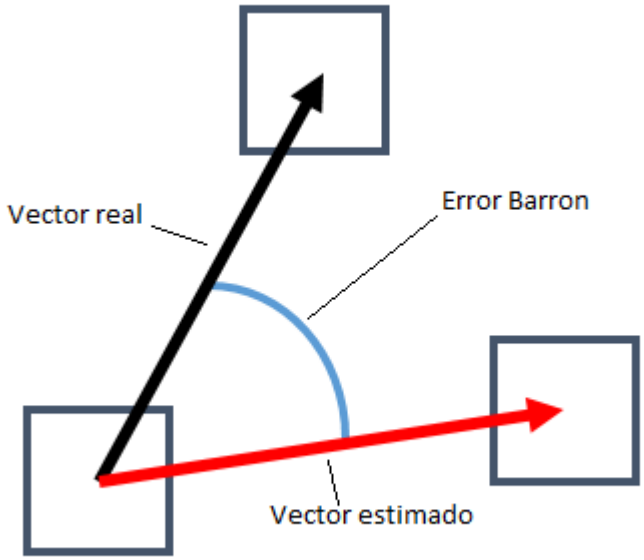

Figura 3.7: *Demostración del tipo de error que Barron pretende calcular.*

Como se puede ver en la Figura 3.7, el error se representa como el ángulo (normalmente en radianes) que forma el vector real con el estimado. Este error nos vendrá bien con estí-

mulos del tipo de *DivergingTree*, ya que al ser un estímulo de zoom, los *pixeles* se moveran describiendo movimientos no paralelos a ninguno de los ejes.

3.5.2. Error de Otte&Nagel (Endpoint Error)

Se describe como el ángulo entre los vectores normalizados $(u, v, 1)$ y $(ur, vr, 1)$ del espacio 3D. Los resultados están normalizados lo cual provoca que los errores que sean tanto desplazamientos cortos como largos tengan la misma influencia en el resultado final. Este error se computa de acuerdo a la siguiente ecuación:

$$E_{endpoint} = \sqrt[2]{(u - u_{gt})^2 + (v - v_{gt})^2} \tag{3.19}$$

Donde u_{gt} y v_{gt} son los valores reales del vector.
De esta manera, podemos obtener la diferencia entre el vector estimado y el real mediante una diferencia de sus componentes hayando, no el ángulo diferencia entre ellos, si no la distancia como se ve en la Figura 3.8

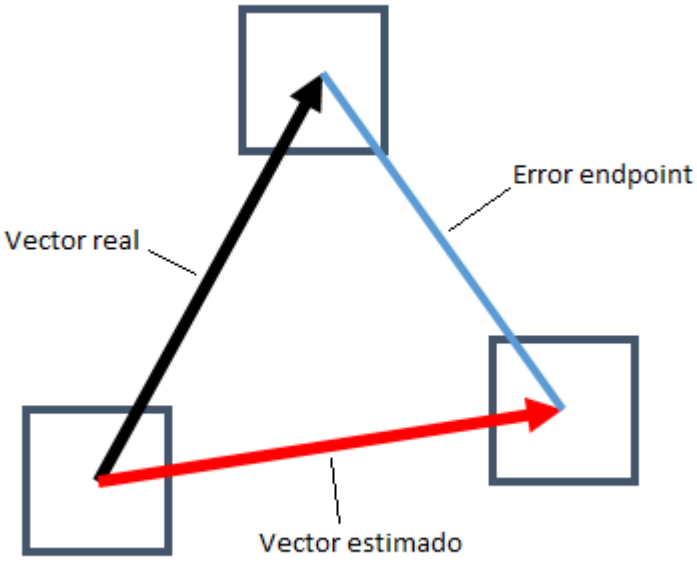

Figura 3.8: *Demostración del tipo de error que Otte&Nagel pretende calcular.*

Capítulo 4

Optimización de algoritmos de flujo óptico

A continuación, explicaremos la implementación realizada sobre los algoritmos de flujo óptico elegidos: **Lucas&Kanade** y **Horn&Schunck**. Se han escogidos estos algoritmos porque han sido objeto de numerosos estudios y porque, aún siendo los dos del modelos de gradiente, son muy distintos entre si, y el paralelismo explotable difiere mucho entre ambos.

Se mostrará, primero, un pequeño apunte que introducirá al buen entendimiento de la implementación, clasificando las versiones del algoritmo a cada cuál más optimizada. Además se explicarán también los cambios que se hicieron para la portabilidad de una plataforma a otra, que es la característica principal de **OpenCL**.

4.1. Implementación común

Ambos algoritmos se han implementado en *C* en lugar de *Fortran* por estar más familiarizados con el primero. Estos algoritmos permitirán una entrada de secuencias de imágenes que serán procesadas de N *frames* en N *frames*, consiguiendo de esta manera más paralelismo a costa de gasto en recursos.

Mediante linea de comandos, los algoritmos permitirán cambiar entre su forma base (sin

optimización) a su forma más optimizada, pasando obviamente por los puntos intermedios, además de permitir también otras opciones como la elección del tipo de datos y el muestreo de tiempos y cálculos de error.

4.2. Implementación Lucas&Kanade

Todo lo explicado en este punto será referido al algoritmo secuencial, para entender antes de pasar a la implementación en **OpenCL** como se estructura el programa.

En el Algoritmo 9 se detalla, en formato de pseudocódigo el esquema general del algoritmo. Se compone de un gran bucle for que va recorriendo todos los *frames* del flujo de imágenes y que encierra los siguientes elementos: punteros a los *frames* de la película que va a procesar y el conjunto de llamadas a las funciones necesarias que pasaremos a explicar a continuación.

En la primera fase de procesamiento del programa, se calculan las derivadas espaciales para los ejes X e Y y, a continuación, se calculan las derivadas temporales. Por último, se calcularán los sumatorios de los productos de las derivadas anteriores para formar las matrices que nos servirán para resolver el sistemas.

Algorithm 9 Implementación Lucas&Kanade.

```
1:  Ux,Vy = metodoLucasKanade(pelicula, filtroX, filtroY, Nfilas, Ncols, Nframes)
2:  for z=0; z<Nframes-1; z++ do
3:      A0 = pelicula + (Nfilas*Ncols)*z
4:      A1 = pelicula + (Nfilas*Ncols)*(z+1)
5:      Ix = calcularDerivadaEspacialX(A0, filtroX)
6:      Iy = calcularDerivadaEspacialY(A0, filtroY)
7:      It = calcularDerivadaTemporal(frames)
8:      sum_Ix2 = calcularSumatorio(Ix, Ix)
9:      sum_Iy2 = calcularSumatorio(Iy, Iy)
10:     sum_IxIy = calcularSumatorio(Ix, Iy)
11:     sum_IxIt = calcularSumatorio(Ix, It)
12:     sum_IyIt = calcularSumatorio(Iy, It)
13:     Ux, Vy = resolverSistema(sum_Ix2, sum_Iy2, sum_IxIy, sum_IxIt, sum_IyIt)
14: end for
```

En el cálculo de las derivadas espaciales se hará uso de filtros. Estos, a su vez, podrán modificar su tamaño para probar diferentes configuraciones con la intención de obtener mejores resultados. A continuación, se muestra el algoritmo que muestra la convolución llevada a cabo para el cálculo de la derivada espacial en el eje X. Para el eje Y el procedimiento es similar pero usando un filtro vertical. Al final, se pasaría de una convolución por columnas a una por filas.

Algorithm 10 Implementación cálculo derivada espacial X.

```
1:  Ix = calcularDerivadaEspacialX(frameAct, filtroX, Nfilas, Ncols, tamFiltro)
2:  for i = 0; i <Nfilas; i++ do
3:     for j = 0; j <Ncols; j++ do
4:        tmp = 0
5:        for k = 0; k <tamFiltro; k++ do
6:           posicion = j + k - (tamFiltro/2)
7:           if posicion >= 0 and posicion <Ncols then
8:              tmp += frameAct[i*Ncols + posicion]*filtroX[k]
9:           end if
10:       end for
11:       Ix[i*Ncols + j] = tmp
12:    end for
13: end for
```

Como resultado de este procedimiento obtendremos la derivada espacial en el eje X de un *frame* dado. Se realiza un recorrido por columnas y filas y a cada *pixel* se le aplica el filtro centrado (para ello, se obliga que este sea impar). Para los casos especiales de los bordes se comprobará primero que el filtro no salga de los limites del *frame*. Esto último se explicará mejor en la implementación de **OpenCL**.

Ahora, pasaremos a mostrar el cálculo de la derivada temporal. En este caso, el filtro que se usará será negativo, lo que significa que se calculará la diferencia entre *frames* consecutivos.

Algorithm 11 Implementación cálculo derivada temporal.

```
1: It = calcularDerivadaEspacialX(frames, filtroT, Nfilas, Ncols, tamFiltro)
2: for i = 0; i <Nfilas; i++ do
3:    for j = 0; j <Ncols; j++ do
4:       It[i*Ncols+j] = 0.0
5:       for k = 0; k <tamFiltro; k++ do
6:          It[i*Ncols+j] += frames[(i*Ncols+j)+(Nfilas*Ncols*k)]*filtroT[k]
7:       end for
8:    end for
9: end for
```

Para el cálculo de los productos y los cuadrados de los resultados anteriores se realizará una multiplicación elemento a elemento. Por simplicidad no se incluye este cálculo en la memoria.

Para el sumatorio de los *pixeles* de la vecindad se le asigna un peso a cada *pixel*. Por simplicidad, este valor será de 1 para todos, por lo que simplemente hay que realizar un sumatorio de los *pixeles* vecinos dentro de un marco, que vendrá especificado por el tamaño de bloque (**tamBloqX** y **tamBloqY**). En el algoritmo 5 se muestra el funcionamiento.

Algorithm 12 Implementación cálculo del sumatorio.

```
1: sumaI = calcularSumatorio(frameAct, Nfilas, Ncols, tamBloqX, tamBloqY)
2: for i = 0; i <Nfilas; i++ do
3:    for j = 0; j <Ncols; j++ do
4:       tmp = 0
5:       for k = 0; k <tamBloqY; k++ do
6:          for l = 0; l <tamBloqX; l++ do
7:             tmp += frameAct[(i+k-(tamBloqY/2))*Ncols+(j+l-(tamBloqX/2))]
8:          end for
9:       end for
10:      sumaI[i*Ncols + j] = tmp
11:   end for
12: end for
```

Por último, la resolución del algoritmo nos proporcionará el flujo óptico. Se construye la matriz B con dimensiones $2x2$ y el vector C de longitud 2, que corresponden a las dos

matrices de la ecuación final de la sección anterior. En este algoritmo calcularemos la inversa de la matriz B (en caso de no poder, asumiremos como nulo este punto) y acabaremos con la multiplicación de la matriz inversa de B y el vector C, dando como resultado U y V que contienen las componentes horizontales y verticales respectivamente.

Algorithm 13 Implementación de la resolución del sistema para **Lucas&Kanade**.

```
1: Ux, Vy = resolverSistema(sumaIx2, sumaIy2, sumaIxIy, sumaIxIt, sumaIyIt, Ncols, Nfi-
   las, Ncols)
2: for i = 0; i <Nfilas; i++ do
3:    for j = 0; j <Ncols; j++ do
4:       C = [ {sumaIx2[i*Ncols+j], sumaIxIy[i*Ncols+j]},
         {sumaIxIy[i*Ncols+j], sumaIy2[i*Ncols+j]} ]
5:       B = [-sumaIxIt[i*Ncols+j], -sumaIyIt[i*Ncols+j]
6:       C = calcularInversa2x2(C)
7:       CB = calcularProducto(C,B)
8:       Ux = CB[0]
9:       Vy = CB[1]
10:   end for
11: end for
```

4.2.1. OpenCL. Optimizaciones y transformaciones

La metodología a usar para acelerar el algoritmo ha sido buscar en el código secuencial los potenciales *kernels* pensando ante todo en usar el máximo paralelismo con la menor dependencia de datos. Para este caso, tomaremos como *kernels* todas las funciones de dentro del bucle principal del programa, estos son:

- Calculo de derivada en X.
- Calculo de derivada en Y.
- Calculo de derivada temporal.
- Multiplicación elemento a elemento.
- Cálculo del sumatorio de los vecinos.
- Resolución del algoritmo (*Core*).

El proceso de transformación en *kernel* ha sido divido en dos fases. La primera fase, la básica, ha sido realizada sin tener en consideración ningún tipo de optimización en el acceso a los datos con el uso de memoria local, donde realmente se obtienen mejores resultados. La segunda fase, la local, es habiendo ya realizado la básica, optimizar esta última para permitir el acceso a los datos desde memoria local, con mucha menos latencia, lo que nos permitirá acelerar el algoritmo en mayor medida.

Para la primera fase no habría que tener cuidado con problemas de lecturas o escrituras corruptas por salirnos del rango de la memoria global, ya que cada *pixel* calculará sus resultados de manera absoluta. El problema viene con el uso de la memoria local, ya que para distintos *workitems* el índice que usa para acceder a memoria local puede ser el mismo, de manera que hay que tener cuidado con el acceso relativo a la memoria local. Más adelante se explicará gráficamente como se realizará el acceso a la memoria local.

A continuación se explicarán ambas fases. Para la fase básica se explicarán todos los *kernels* y la implementación que se ha realizado sobre ellos, mientras que para la fase local se explicarán solo aquellos *kernels* que se hayan modificado, que simplemente serán tres y coincidirán con el cálculo de las derivadas y de los sumatorios. Cabe mencionar que se reservará el uso de las variables *i, j* y *k* para representar las posiciones en filas, columnas y *frames*, respectivamente, de los *workitems* basándonos en la gestión que hace **OpenCL** (tal y como se explicaba ya en el apartado de *hardware*).

Fase básica

Para el cálculo de la derivada tanto en eje X como Y el procedimiento es similar. Por acortar, se mostrará simplemente la del eje X, pero es extrapolable al eje Y. En el Algoritmo 14 se muestra el código

Algorithm 14 Implementación cálculo derivada espacial X en *kernel*.

```
1: Ix = imfilterX(movie, filter, filterSize, Nrows, Ncols, Nframes)
2: if i <Nrows and j <Ncols and k <Nframes-1 then
3:    Ix[(i*Ncols+j)+(Ncols*Nrows*k)] = 0;
4:    if j <filterSize/2 then
5:       for m = (filterSize/2)-j; m <filterSize; m++ do
6:          pos = j + m - (filterSize/2)
7:          Ix[(i*Ncols+j)+(Ncols*Nrows*k)] += movie[(i*Ncols+pos)+(Ncols*Nrows*k)]*filtro[m]
8:       end for
9:    else if j >= Ncols-(filterSize/2) then
10:      for m = 0; m <(filterSize/2)+(Ncols-j); m++ do
11:         pos = j + m - (filterSize/2)
12:         Ix[(i*Ncols+j)+(Ncols*Nrows*k)] += movie[(i*Ncols+pos)+(Ncols*Nrows*k)]*filtro[m]
13:      end for
14:   else
15:      for m = 0; m <(filterSize/2); m++ do
16:         pos = j + m - (filterSize/2)
17:         Ix[(i*Ncols+j)+(Ncols*Nrows*k)] += movie[(i*Ncols+pos)+(Ncols*Nrows*k)]*filtro[m]
18:      end for
19:   end if
20: end if
```

Como se puede observar en el algoritmo, se lleva a cabo una selección de casos dependiendo de los valores en j, es decir, en columnas del *workitem* que ejecuta el código. Estos casos dependen del tamaño de filtro, ya que debemos preocuparnos de no aplicar el filtro a valores de fuera del rango de la película.

En la Figura 4.7 se puede ver como se dividen las distintas zonas de acción para el caso de un tamaño de filtro 5. En rojo viene representada la zona de la imagen en la que hay algún valor del filtro que no se calcule por salirse fuera del rango. Este cálculo viene representado en la imagen en color verde claro, siendo el verde oscuro el *pixel* sobre el cual se guarda el valor del filtro. No es hasta que salimos de la zona roja ($TamFiltro/2$) que calculamos el filtro sin restricción.

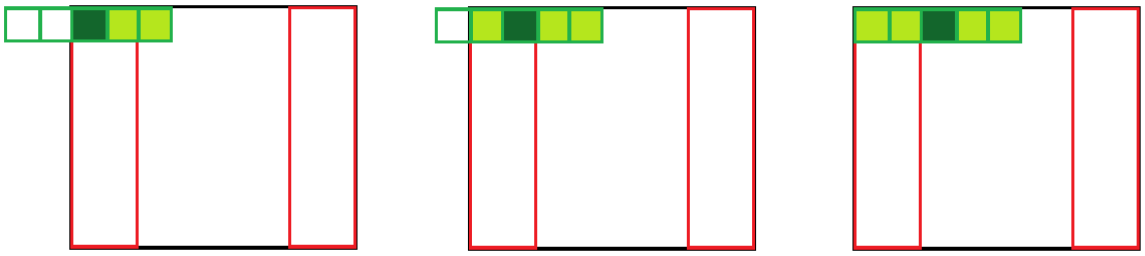

Figura 4.1: *Diagrama de actuación en el cálculo de la derivada*

Para el cálculo de la derivada temporal y de las multiplicaciones y cuadrados necesarios para obtener el sistema de ecuaciones usamos dos *kernels*, el de resta y multiplicación de matrices elemento a elemento. Nuevamente, por simplificar la memoria, no se explicará la transformación a *kernel* de estas funciones, sin embargo, se pasará directamente a explicar la transformación en *kernel* del cálculo de la vecindad.

Para este cálculo se tienen que tener encuentra las mismas restricciones que para la derivada en X y en Y juntas, es decir, que el filtro (o en este caso el marco de vecindad) no salga ni por el norte ni el sur, ni por el este ni el oeste; en caso de que esto ocurriese, no se debería aplicar la suma para esos *pixeles*. En nuestro caso, para aquellos *pixeles* para los que n haya que actuar se tomará el valor cero, haciendo al final un sumatorio de todos los valores sin preocuparnos de salirnos o no.

En el Algoritmo 15 se expone el código que calcula el sumatorio. Se ha dividido el *kernel* en 3 partes dependiendo del posicionamiento del *pixel* a ejecutar, esto servirá para que la explicación sea más sencilla y se entienda mejor.

Algorithm 15 Implementación cálculo del sumatorio de vecindad en kernel **(Parte 1)**.

```
1: Sum = imfilterS(movie, filter, BX, BY, Nrows, Ncols, Nframes)
2: if i <Nrows and j <Ncols and k <Nframes-1 then
3:    tmp = 0;
4:    if i <(BY/2) and j <(BX/2) then
5:       for m = (BY/2)-i; m <BY; m++ do
6:          for n = (BX/2)-j; n <BX; n++ do
7:             posX = j+n-(BX/2); posY = i+m-(BY/2)
8:             tmp += movie[(posY*Ncols+posX) + (Ncols*Nrows*k)]
9:          end for
10:      end for
11:   else if i >= Nrows-(BY/2) and j >= Ncols-(BX/2) then
12:      for m=0; m<(BY/2)+(Nrows-i); m++ do
13:         for n=0; n<(BX/2)+(Ncols-j); n++ do
14:            posX = j+n-(BX/2); posY = i+m-(BY/2)
15:            tmp += movie[(posY*Ncols+posX) + (Ncols*Nrows*k)]
16:         end for
17:      end for
18:   else if j>=Ncols-(BX/2) and i<(BY/2) then
19:      for m=(BY/2)-i; m<BY; m++ do
20:         for n=0; n<(BX/2)+(Ncols-j); n++ do
21:            posX = j+n-(BX/2); posY = i+m-(BY/2)
22:            tmp += movie[(posY*Ncols+posX) + (Ncols*Nrows*k)]
23:         end for
24:      end for
25:   else if i>=Nrows-(BY/2) and j<(BX/2) then
26:      for m=0; m<(BY/2)+(Nrows-i); m++ do
27:         for n=(BX/2)-j; n<BX; n++ do
28:            posX = j+n-(BX/2); posY = i+m-(BY/2)
29:            tmp += movie[(posY*Ncols+posX) + (Ncols*Nrows*k)]
30:         end for
31:      end for
32:   end if
33: end if
```

En esta primera parte se calculan aquellos *workitems* que hacen el cálculo de los *pixels* que se encuentran dentro del limite de cálculo de vecindad, es decir, aquellos *pixeles* que no dependen de restricciones de lecturas fuera de rango como se explicó en la Figura 4.7. Más concretamente, calcula el sumatorio de los *pixeles* del norte, sur, este y oeste de la imagen (obviando las esquinas), dejando un margen de $(BY/2)$ o $(BX/2)$ (dependiendo de si es

norte y sur o este y oeste).

Esto se ilustra en la Figura 4.2. En rojo se muestra la zona afectada y el verde el bloque de vecindad, al igual que se hizo en la Figura 4.7. En este caso también es para una vecindad de 5.

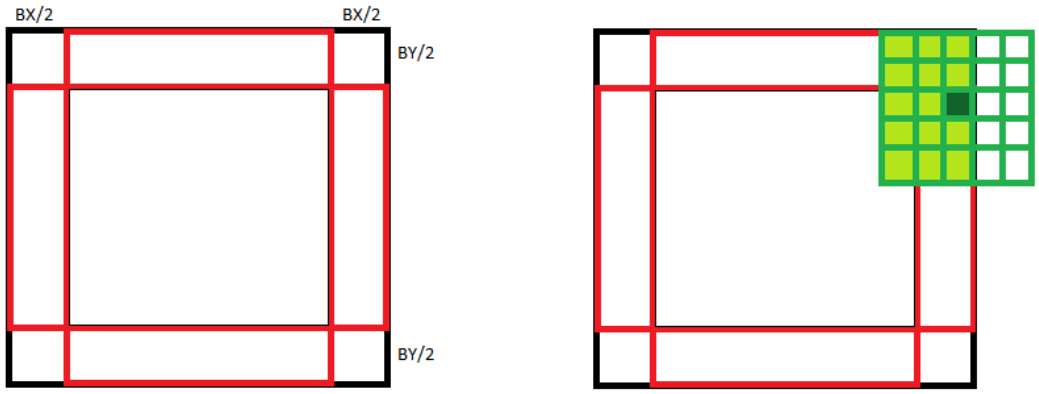

Figura 4.2: *Diagrama de actuación en el cálculo del sumatorio* ***(PARTE 1)***

En la siguiente parte del código tenemos el cálculo del sumatorio para las esquinas de la imagen. El tratamiento de las condiciones es muy similar al anterior pero, en este caso, se tratarán como combinaciones lineales de los anteriores. Por ejemplo, si queremos calcular la esquina superior derecha hay que combinar el condicionamiento de la zona norte con el de la zona oeste. Ampliando esto a las 4 esquinas obtenemos el código del Algoritmo 16

Algorithm 16 Implementación cálculo del sumatorio de vecindad en kernel **(Parte 2)**.

```
1: if j<(BX/2) and (i>=(BY/2) and i<Nrows-(BY/2)) then
2:    for m=0; m<BY; m++ do
3:       for n=(BX/2)-j; n<BX; n++ do
4:          posX = j+n-(BX/2)
5:          posY = i+m-(BY/2)
6:          tmp += movie[(posY*Ncols+posX) + (Ncols*Nrows*k)]
7:       end for
8:    end for
9: else if (j<Ncols-(BX/2) and j>=(BX/2)) and i<(BY/2) then
10:   for m=(BY/2)-i; m<BY; m++ do
11:      for n=0; n<BX; n++ do
12:         posX = j+n-(BX/2)
13:         posY = i+m-(BY/2)
14:         tmp += movie[(posY*Ncols+posX) + (Ncols*Nrows*k)]
15:      end for
16:   end for
17: else if j>=Ncols-(BX/2) and (i>=(BY/2) and i<Nrows-(BY/2)) then
18:   for m=0; m<BY; m++ do
19:      for n=0; n<(BX/2)+(Ncols-j); n++ do
20:         posX = j+n-(BX/2)
21:         posY = i+m-(BY/2)
22:         tmp += movie[(posY*Ncols+posX) + (Ncols*Nrows*k)]
23:      end for
24:   end for
25: else if (j<Ncols-(BX/2) and j>=(BX/2)) and i>=Nrows-dify then
26:   for m=0; m<(BY/2)+(Nrows-i); m++ do
27:      for n=0; n<BX; n++ do
28:         posX = j+n-(BX/2)
29:         posY = i+m-(BY/2)
30:         tmp += movie[(posY*Ncols+posX) + (Ncols*Nrows*k)]
31:      end for
32:   end for
33: end if
```

El funcionamiento es igual que en la parte 1 pero cambiando el precondicionamiento. A continuación, en la Figura 4.3, se expone una imagen explicativa de como funciona esta segunda parte y de como se calcula el sumatorio para un *pixel* central, representado en verde oscuro.

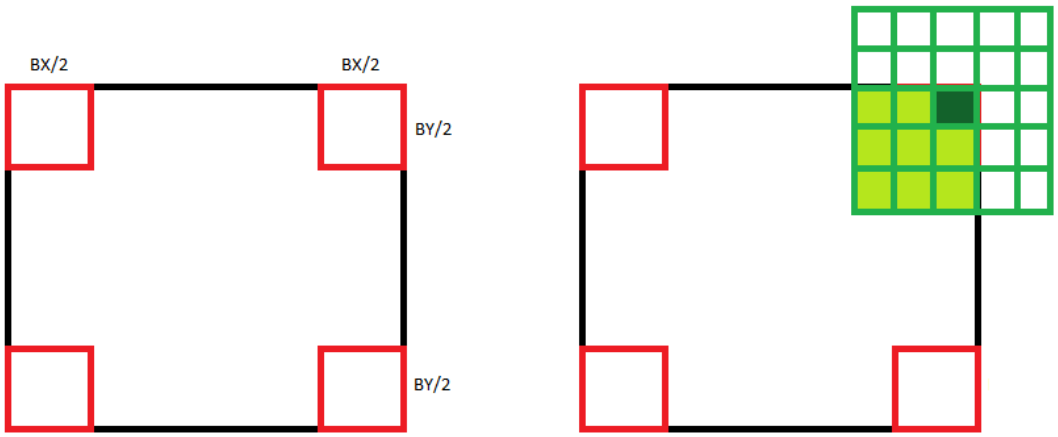

Figura 4.3: *Diagrama de actuación en el cálculo del sumatorio* ***(PARTE 2)***

Por último, la parte 3 del algoritmo que se encarga de calcular el sumatorio de la vecindad para el resto de *pixeles* de la imagen. Se denomina la parte central y no tiene restricciones en cuanto a lecturas fuera de rango de la imagen, por lo tanto, serán los únicos *pixeles* que calculen el sumatorio de todos los *pixeles* de la vecindad. En el Algoritmo 17 se muestra el resto del código para la función *imfilterS*.

Algorithm 17 Implementación cálculo del sumatorio de vecindad en kernel **(Parte 3)**.

```
1: if (i >= (BY/2) and i <Nrows-(BY/2)) and (j >= (BX/2) and j <Ncols-(BX/2))
   then
2:    for m=0; m<BY; m++ do
3:       for n=0; n<BX; n++ do
4:          posX = j+n-(BX/2)
5:          posY = i+m-(BY/2)
6:          tmp += movie[(posY*Ncols+posX) + (Ncols*Nrows*k)]
7:       end for
8:    end for
9:    Sum[(i*Ncols+j)+(Ncols*Nrows*k)] = tmp;
10: end if
```

Como ya es común, en la Figura 4.4 se puede ver el rango de actuación de la tercera parte del algoritmo. Como observación final, se puede apreciar que solo para este caso todos

los recuadros de la vecindad se tornan verdes claros. Esto es debido a que todos están dentro del rango de la imagen.

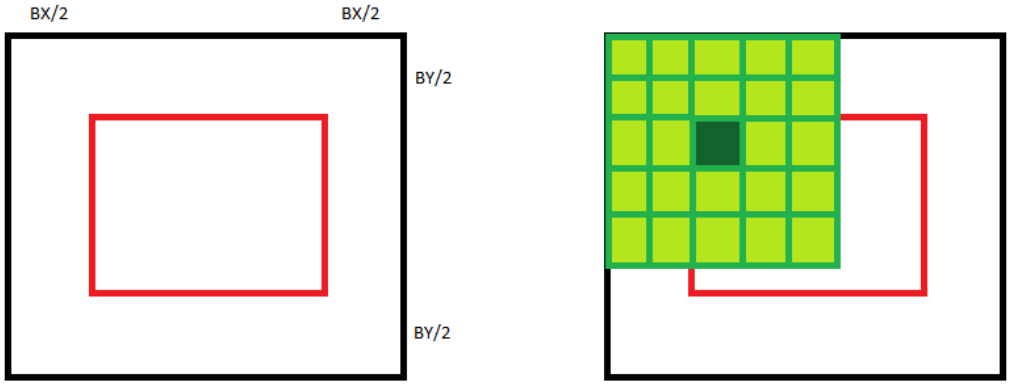

Figura 4.4: *Diagrama de actuación en el cálculo del sumatorio* ***(PARTE 3)***

Para acabar esta sección explicaremos la implementación de la resolución del sistema de ecuaciones resultante. Esta implementación tendrá el nombre de *Core* (ya que es el núcleo de **Lucas&Kanade**) y devolverá el flujo óptico tal y como especificamos al principio del apartado.

Fase local

Como ya se advirtió con anterioridad, solo serán trasladados a uso de memoria local de **OpenCL** aquellos *kernels* que tengan un gran número de acceso a la misma dirección de memoria global (alta localidad temporal). Teniendo en cuenta esta restricción, los *kernels* que sufrirán el cambio serán los de filtrado, tanto para X como para Y como para el sumatorio, ya que son estos los que necesitan aplicar su matriz de filtrado a varias posición en la misma iteración. Para más ayuda, es aconsejable revisar las figuras del punto anterior.

Se comenzará explicando los cambios realizados para el filtro en X. No se explicará el filtrado en Y por ser homologo al de X.

Para optimizar el tiempo y que de esta manera nuestro algoritmo sea más eficiente se ha tenido en cuenta el problema que originan los saltos a la hora de compilar el programa para el tiempo de ejecución. Sabiendo que es más eficiente un código secuencial que un bucle, se ha escrito todo el código en secuencia aprovechándonos de que el tamaño del filtro ha sido estipulado (yendo de 3 a 13).

Algorithm 18 Implementación cálculo del filtro en X con memoria local.

```
1:  if i <Nrows and j <Ncols and k <Nframes-1 then
2:     switch (filterSize)
3:     case 3:
4:        if j <(FS/2) then
5:           tmp += sharedA[(iL*Lcols+(jL+(FS/2)) )+(Lcols*Lrows*kL)]*sharedB[1];
6:           tmp += sharedA[(iL*Lcols+(jL+(FS/2))+1)+(Lcols*Lrows*kL)]*sharedB[2];
7:        else if j <(Ncols-1-(FS/2)) then
8:           tmp += sharedA[(iL*Lcols+(jL+(FS/2))-1)+(Lcols*Lrows*kL)]*sharedB[0];
9:           tmp += sharedA[(iL*Lcols+(jL+(FS/2)) )+(Lcols*Lrows*kL)]*sharedB[1];
10:       else
11:          tmp += sharedA[(iL*Lcols+(jL+(FS/2))-1)+(Lcols*Lrows*kL)]*sharedB[0];
12:          tmp += sharedA[(iL*Lcols+(jL+(FS/2)) )+(Lcols*Lrows*kL)]*sharedB[1];
13:          tmp += sharedA[(iL*Lcols+(jL+(FS/2))+1)+(Lcols*Lrows*kL)]*sharedB[2];
14:       end if
15:    end switch
16:    Ix[(i*Ncols+j)+(Ncols*Nrows*k)] = tmp;
17: end if
```

Usaremos el código del algoritmo 18 para explicar lo que supone la abstracción de la memoria local. La matriz de acceso local llamada *sharedB* es la encargada de albergar el filtro, mientras que la denominada *sharedA* es la encargada de mantener la información global a nivel de bloque, es decir, la memoria local contiene, no solo aquellos pixeles que se suponen irían dentro de un mismo *workgroup* de OpenCL, si no que también almacena aquellos pixeles delimitadores que son necesarios para el cálculo del filtro. En la Figura 4.5 se puede observar el hecho explicado.

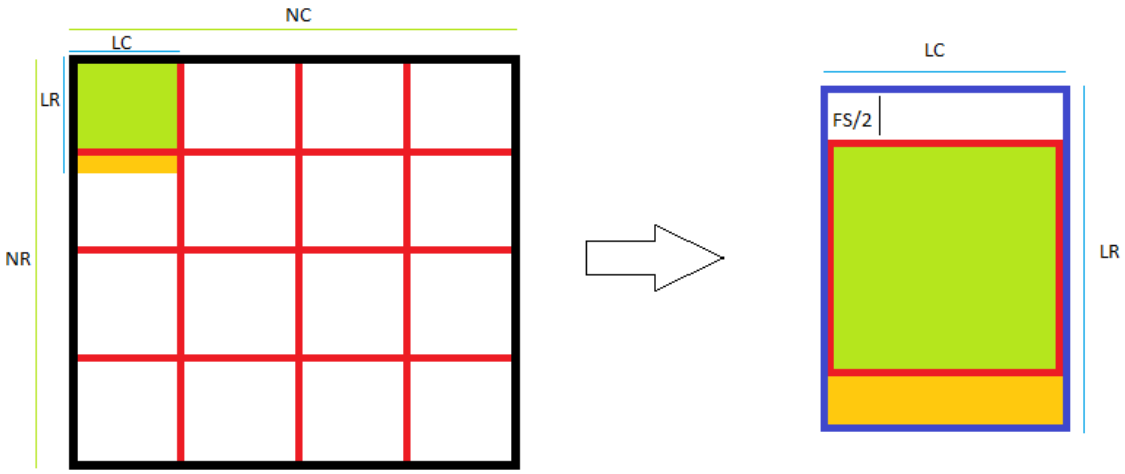

Figura 4.5: *Uso de la memoria local para almacenar el workgroup en el cálculo del filtro*

Al tratarse de una matriz de tamaño relativo al *workgroup* se usarán las directivas *get_local_id()* de OpenCL para, además de obtener el valor de x, y y z de un *workitem* a nivel global, podamos obtener los mismo valores a nivel local y de esta manera acceder mediante dos índices a diversos niveles de memoria.

Se puede observar como al acceder a la matriz *sharedA* se hace con un desplazamiento *jL + (FS/2)*. Este deplazamiento es necesario para mantener la homogeneidad estructural de la memoria global dentro de la memoria local como bien se especificaba en la Figura 4.5. Mediante unas lineas de código antes de la ejecución del algoritmo se inicializa la matriz teniendo en cuenta este desplazamiento. En el Algoritmo 19 se puede entender este proceso.

Algorithm 19 Inicialización de las matrices en memoria local.

```
1: sharedA[iL*Lcols+(jL+(FS/2))+(Lcols*Lrows*kL)] = 0.0;
2: if iL <FS and jL == 0 then
3:    sharedB[iL] = B[iL];
4: end if
5: if i <Nrows and j <Ncols and k <Nframes-1 then
6:    sharedA[(iL*Lcols+(jL+(FS/2)))+(Lcols*Lrows*kL)] =
      A[(i*Ncols+j)+(Ncols*Nrows*k)]; {Aqui calculamos las condiciones de contorno}
7:    if jL >Lcols-1-(FS/2)*3 then
8:       sharedA[(iL*Lcols+(jL+(FS/2))+(FS/2))+(Lcols*Lrows*kL)] = (j >Ncols-1-
         (FS/2)) ? 0.0 : A[(i*Ncols+(j+(FS/2)))+(Ncols*Nrows*k)];
9:    else if jL <(FS/2) then
10:      sharedA[(iL*Lcols+jL)+(Lcols*Lrows*kL)] = (j <(FS/2)) ? 0.0 : A[(i*Ncols+(j-
         (FS/2)))+(Ncols*Nrows*k)];
11:   end if
12: end if
13: BARRIER
```

Al final de la inicialización colocaremos una barrera para no permitir que el algoritmo empiece sin tener todo almacenado en nivel local.

Para finalizar con Lucas&Kanade, vamos a explicar la transformación a memoria local del cálculo del sumatorio de los vecinos.

Este *kernel* lleva un tratamiento especial, ya que es imposible realizar este sin saber el tamaño de la vecindad y, por lo tanto, esta variable se tiene que tener en cuenta en el cálculo del invariente que resultará en un bucle que recorrerá, para cada píxel de la imagen, todos aquellos pixeles vecinos a una distancia de *(FS/2)*.

Al igual que se hace en la versión base, se realiza una distinción en cuanto a la posición del píxel en la imagen, obteniendo los 8 casos mencionados en la implementación con memoria global. Esta vez, estos casos se usarán para inicializar la matriz *sharedA* de manera que, después en el bucle del cálculo del sumatorio, no se tengan que hacer comprobación de lecturas fuera de rango.

Algorithm 20 Inicialización de las matrices en memoria local **(Bordes)**.

```
1: sharedA[(iL+(BY/2))*Lcols+(jL+(BX/2))+(Lcols*Lrows*kL)] = 0.0;
2: if i <Nrows and j <Ncols and k <nFrames-1 then
3:     sharedA[((iL+(BY/2))*Lcols+(jL+(BX/2)))+(Lcols*Lrows*kL)] =
       A[(i*Ncols+j)+(Ncols*Nrows*k)];
4: end if{//Norte}
5: if iL <(BY/2) then
6:     sharedA[(iL*Lcols)+(jL+(BX/2)) + (kL*Lcols*Lrows)] = (i <(BY/2) or j >Ncols-1)
       ? 0.0 : A[((i-(BY/2))*Ncols)+ j + (Ncols*Nrows*k)];
7: end if{//Sur}
8: if iL >Lrows - 1 - (BY/2)*3 then
9:     sharedA[((iL+(BY/2)*2)*Lcols)+(jL+(BX/2)) + (kL*Lcols*Lrows)] = (i >Nrows-1
       - (BY/2) or j >Ncols-1) ? 0.0 : A[((i+(BY/2))*Ncols)+ j + (Ncols*Nrows*k)];
10: end if{//Oeste}
11: if jL <(BX/2) then
12:     sharedA[((iL+(BY/2))*Lcols)+ jL + (Lcols*Lrows*kL)] = (j <(BX/2) or i >Nrows-
        1) ? 0.0 : A[(i*Ncols)+(j-(BX/2))+(Ncols*Nrows*k)];
13: end if{//Este}
14: if jL >Lcols - 1 - (BX/2)*3 then
15:     sharedA[((iL+(BY/2))*Lcols) + (jL + (BX/2)*2) + (Lcols*Lrows*kL)] = (j >Ncols-
        1-(BX/2) or i >Nrows-1) ? 0.0 : A[(i*Ncols)+(j+(BX/2))+(Ncols*Nrows*k)];
16: end if
```

Algorithm 21 Inicialización de las matrices en memoria local **(Esquinas)**.

```
1: if iL <(BY/2) and jL <(BX/2) then
2:     sharedA[iL*Lcols+jL + (kL*Lcols*Lrows)] = (i <(BY/2) or j <(BX/2)) ? 0.0 :
       A[(((i-(BY/2))*Ncols)+ (j-(BX/2) + (Ncols*Nrows*k)];
3: end if{//Arriba izquierda}
4: if iL <(BY/2) and jL >Lcols - 1 - (BX/2)*3 then
5:     sharedA[(iL*Lcols)+(jL+(BX/2)*2) + (kL*Lcols*Lrows)] = (i <(BY/2) or j >Ncols
       - 1 - (BX/2)) ? 0.0 : A[((i-(BY/2))*Ncols)+ (j + (BX/2)) + (Ncols*Nrows*k)];
6: end if{//Arriba derecha}
7: if iL >Lrows - 1 - (BY/2)*3 and jL <(BX/2) then
8:     sharedA[((iL+(BY/2)*2)*Lcols)+ jL + (Lcols*Lrows*kL)] = (j <(BX/2) or i
       >Nrows-1-(BY/2)) ? 0.0 : A[((i+(BY/2)*Ncols)+(j-(BX/2))+(Ncols*Nrows*k)];
9: end if{//Abajo izquierda}
10: if iL >Lrows - 1 - (BY/2)*3 and jL >Lcols - 1 - (BX/2)*3 then
11:     sharedA[((iL+(BY/2)*2)*Lcols) + (jL + (BX/2)*2) + (Lcols*Lrows*kL)]
        = (j >Ncols-1-(BX/2) or i >Nrows-1-(BY/2)) ? 0.0 :
        A[((i+(BY/2))*Ncols)+(j+(BX/2))+(Ncols*Nrows*k)];
12: end if{//Abajo derecha}
```

En el Algoritmo 20 y Algoritmo 21 se inicializan a 0 aquellas posiciones de la matriz local que no deberían ser accedidas. Así se conseguiría que, en el bucle del final, no se tuviera que tener cuidado al hacer el sumatorio, haciendo que la dura tarea de generar un salto óptimo del compilador no sea tan costosa. En el primer algoritmo mencionado se realiza la inicialización de los bordes de la imagen sin tener en cuanta las esquinas, así, cada *pixel* calculará su posición y las *(B/2)* siguientes. En el segundo algoritmo mencionado se hace el cálculo de las esquinas, combinando los dos principios de X e Y.

Algorithm 22 Inicialización de las matrices en memoria local.

```
1: for n <= (BY/2) do
2:    for m <= (BX/2) do
3:       tmp+=sharedA[((iL+(BY/2)+n)*Lcols)+(jL+(BX/2)+m)+(Lcols*Lrows*kL)];
4:    end for
5: end for
6: R[(i*Ncols+j)+(Ncols*Nrows*k)] = tmp;
```

Por último en el Algoritmo 22 se almacena la información del sumatorio para guardarla en la matriz de salida más tarde.

El mismo principio usado en el cálculo de la derivada en X para inicializar la matriz *sharedA*, es ahora usado aquí también, con la diferencia de que en este caso se hará lo mismo tanto para el eje X como para el eje Y.

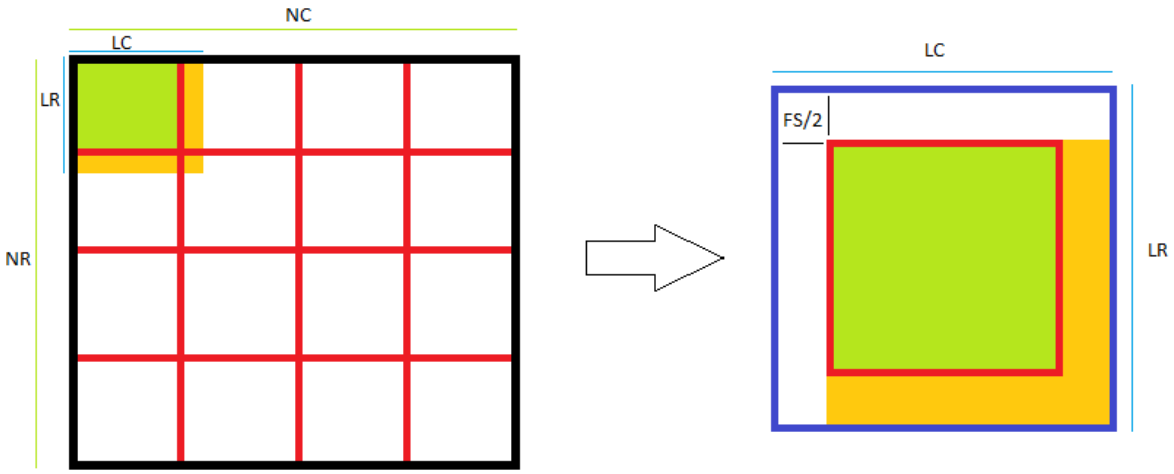

Figura 4.6: *Uso de la memoria local para almacenar el workgroup en el cálculo de la vecindad*

Gracias a la Figura 4.6 se puede ver claramente como lo dicho anteriormente se cumple.

4.3. Implementación Horn&Schunk

4.3.1. Código secuencial

Tal y como se hizo en **Lucas&Kanade** procederemos primero a explicar el código secuencial para una mejor visualización de la estructura del programa.

El algoritmo de **Horn&Schunk** consiste como cualquier otro algoritmo de flujo óptico en computar la dirección y velocidad del movimiento de los *pixeles*, el cual guarda en las variables u y v. Este algoritmo tiene varios pasos, el primer paso consiste en realizar un *mipmaping*[1] de los dos *frames* sobre los cuales realizar el flujo óptico. En este algoritmo seleccionamos por parámetro el número de niveles que queremos aplicar a la imagen.

En este código realizamos n veces, donde n es el número de niveles que deseamos efectuar, el cálculo de las dimensiones del siguiente nivel del *mipmaping* y el propio proceso de *mipmaping* con la función **Downscale** la cual veremos con más detalle cuando veamos la implementación de **OpenCL** más adelante.

[1] El proceso de *mipmaping* consiste en dividir las dimensiones de una imagen por 2 hasta que no pueda dividirse más. Cada vez que la imagen se divide por 2 conforma un nivel.

Algorithm 23 Bajada en la piramide de niveles.

```
1: for ; currentLevel >0; –currentLevel do
2:     int nw = pW[currentLevel] / 2
3:     int nh = pH[currentLevel] / 2
4:     int ns = iAlignUp(nw)
5:     pI0[currentLevel - 1] = pI0_tmp + p_pI_tmp
6:     pI1[currentLevel - 1] = pI1_tmp + p_pI_tmp
7:     p_pI_tmp += ns * nh
8:     Downscale(pI0[currentLevel], pW[currentLevel], pH[currentLevel], pS[currentLevel],
       nw, nh, ns, (float *)pI0[currentLevel - 1])
9:     Downscale(pI1[currentLevel], pW[currentLevel], pH[currentLevel], pS[currentLevel],
       nw, nh, ns, (float *)pI1[currentLevel - 1])
10:    pW[currentLevel - 1] = nw;
11:    pH[currentLevel - 1] = nh;
12:    pS[currentLevel - 1] = ns;
13: end for
```

Después del proceso de *mipmaping* realizado anteriormente se realiza el proceso de calculo ascendente del flujo óptico, por lo que nos encontramos con la columna vertebral del algoritmo de **Horn&Schunk**. Durante este proceso inicializamos todas las componentes de las variables de u y v a 0, seguido a esto vamos del nivel 0 al nivel $nLevels - 1$ (que es la variable que indica el número de niveles pedido por el usuario) realizando las iteraciones sobre los datos que veremos en el algoritmo siguiente. Cuando en cada nivel hemos realizado las operaciones oportunas miramos si todavía quedan niveles por los que subir hacia arriba, en caso afirmativo lo que hacemos es el proceso contrario al desarrollado en el *mipmaping*. Cogemos la imagen de un cierto nivel i y calculamos el índice de escalado que tenemos que aplicarle a la imagen para pasarla a las dimensiones del nivel superior, actualizando a la misma vez la u y la v, todo esto se realiza mediante la llamada a la función **Upscale**. Una vez realizado este proceso para los dos *frames* intercambiamos los valores de las variables u y nu y de las variables v y nv.

Algorithm 24 Bucle principal de subida de niveles.

```
1:  memset(u, 0, stride * height * sizeof(float));
2:  memset(v, 0, stride * height * sizeof(float));
3:  for ; currentLevel <nLevels; ++currentLevel do
4:     Codigo del algoritmo 23.
5:     if currentLevel != nLevels - 1 then
6:        float scaleX = (float)pW[currentLevel + 1]/(float)pW[currentLevel];
7:        Upscale(u, pW[currentLevel], pH[currentLevel], pS[currentLevel], pW[currentLevel
          + 1], pH[currentLevel + 1], pS[currentLevel + 1], scaleX, nu);
8:        float scaleY = (float)pH[currentLevel + 1]/(float)pH[currentLevel];
9:        Upscale(v, pW[currentLevel], pH[currentLevel], pS[currentLevel], pW[currentLevel
          + 1], pH[currentLevel + 1], pS[currentLevel + 1], scaleY, nv);
10:       Swap(u, nu);
11:       Swap(v, nv);
12:    end if
13: end for
```

Como se adelantó antes, durante la subida por los niveles se realiza un cálculo necesario antes de pasar al nivel superior. Este calculo se compone de un bucle que se repetirá *nWarpIters* repeticiones. Dentro de este bucle realizamos una inicialización de las variables $du0, dv0, du1, dv1$. Después de esto, realizamos una operación llamada *deformación de la imagen* mediante la llamada a la función **WarpImage** que realiza unas ciertas operaciones que se verán a pleno detalle en el apartado del código **OpenCL**. Tras esto se realiza una llamada donde se calculan las derivadas espaciales y temporales con la llamada al código de **computeDerivatives**, tras esto se realiza *nSolveIters* iteraciones en las que ejecuta el método **Jacobi** para todos los *píxeles* de la imagen y guarda los resultados en $du1$ y $dv1$. Al final de cada iteración reconecta las variables de salida de ese método con las de entrada, de ahí que se intercambien los valores de $du0$ con $du1$ y $dv0$ con $dv1$. Finalmente al final de cada iteración del bucle exterior se suman los valores de u y v con la salida del ultimo **SolveForUpdate** ejecutado es decir, suma u con $du0$ y v con $dv0$.

Algorithm 25 Bucle de operaciones de contorno.

```
1:  for int warpIter = 0; warpIter <nWarpIters; ++warpIter do
2:    memset(du0, 0, pixelCountAligned * sizeof(float));
3:    memset(dv0, 0, pixelCountAligned * sizeof(float));
4:    memset(du1, 0, pixelCountAligned * sizeof(float));
5:    memset(dv1, 0, pixelCountAligned * sizeof(float));
6:    WarpImage(pI1[currentLevel], pW[currentLevel], pH[currentLevel], pS[currentLevel],
      u, v, tmp);
7:    ComputeDerivatives(pI0[currentLevel], tmp, pW[currentLevel], pH[currentLevel],
      pS[currentLevel], Ix, Iy, Iz);
8:    for int iter = 0; iter <nSolverIters; ++iter do
9:      SolveForUpdate(du0, dv0, Ix, Iy, Iz, pW[currentLevel], pH[currentLevel],
        pS[currentLevel], alpha, du1, dv1);
10:     Swap(du0, du1);
11:     Swap(dv0, dv1);
12:   end for
13:   for int i = 0; i <pH[currentLevel] * pS[currentLevel]; ++i do
14:     u[i] += du0[i];
15:     v[i] += dv0[i];
16:   end for
17: end for
```

4.3.2. Código acelerado con OpenCL

Una vez que ya hemos entendido lo que hace el código procedemos a mostrar las zonas del código las cuales pasamos un poco por alto antes, es aquí cuando entraremos por fin en el terreno del código **OpenCL**.

En primer lugar veremos el código que realiza el *downsampling* por la imagen, como se puede ver en el código el algoritmo coge los *píxeles* adyacentes por cada *píxel* de la imagen y realiza una media aritmética sobre el valor de estos cuatro *píxeles*. Como se puede apreciar se realiza una llamada a la función **Tex2Di_acc**, las operaciones realizadas por esta función son las comentadas durante el capítulo anterior de introducción en el que se presentó teóricamente este algoritmo, realizando el reflejo sobre los índices que queden fuera de rango reflejándolos hacia valores dentro del rango.

Algorithm 26 Código que realiza el downsampling.

```
1: __kernel void downscale_CL(__global float *src, __global float *out, const int width,
   const int height, const int stride, const int offset, const int newWidth, const int new-
   Height, const int newStride, const int newOffset)
2: {
3: int i, j, srcX, srcY, x, y;
4: float sum;
5: i = get_global_id(0);
6: j = get_global_id(1);
7: if i <newHeight && j <newWidth then
8:    srcX = j * 2;
9:    srcY = i * 2;
10:   sum = 0.0f;
11:   sum = Tex2Di_acc(sum, src+offset, width, height, stride, srcX + 0, srcY + 0, x, y);
12:   sum = Tex2Di_acc(sum, src+offset, width, height, stride, srcX + 0, srcY + 1, x, y);
13:   sum = Tex2Di_acc(sum, src+offset, width, height, stride, srcX + 1, srcY + 0, x, y);
14:   sum = Tex2Di_acc(sum, src+offset, width, height, stride, srcX + 1, srcY + 1, x, y);
15:   // normalize
16:   sum *= 0.25f;
17:   out[j + i * newStride + newOffset] = sum;
18: end if
19: }
```

El siguiente código realiza la operación de prolongación explicada en la introducción de este algoritmo en el capítulo anterior, como se puede ver en el código para *píxel* de la imagen del nivel actual se le resta 0.5 y se le multiplica por 0.5, esto es así para poder obtener el *píxel* del nivel anterior el cual por el proceso de prolongación va a acabar dándole información a él.

Con toda esta información computada procede a llamar a la función **tex2D_CL** la cuál veremos a continuación para ver en más detalle las operaciones auxiliares usadas en durante este proceso de prolongación.

Algorithm 27 Bucle de operaciones de contorno.

```
1: __kernel void upscale_CL(__global float *src, __global float *out, const int width,
   const int height, const int stride, const int newWidth, const int newHeight, const int
   newStride, const float scale)
2: {
3: int i, j;
4: float x, y;
5: i = get_global_id(0);
6: j = get_global_id(1);
7: if i <newHeight && j <newWidth then
8:    x = ((float)j - 0.5f) * 0.5f;
9:    y = ((float)i - 0.5f) * 0.5f;
10:   out[j + i * newStride] = tex2D_CL(src, width, height, stride, x, y) * scale;
11: end if
12: }
```

En esta función a partir de la información computada en la anterior desde la cual es llamada esta, lo que hace es dividir la x y la y en parte entera y parte decimal. La parte entera la usará para obtener los índices de los *píxeles* que participarán en el valor devuelto para el *píxel* del nivel superior. La parte decimal en cambio será usada como variable de ajuste del peso que darán estos *píxeles* al valor final del *píxel* del nivel superior.

Una vez producida la división en las primeras líneas de código, continúa verificando si las variables **ix0** e **iy0** están dentro de rango. Continúa generando **ix1** e **iy1** vuelve a verifi-

car para estas si están dentro de rango y finalmente se produce el computo de las operaciones.

El valor que produce el computo de las operaciones es devuelto y multiplicado por un parámetro que indica el índice de escalado que aplicar a los datos y se le asigna al *píxel* de la imagen del siguiente nivel.

Algorithm 28 Bucle de operaciones de contorno.

```
1: float tex2D_CL(__global float *t, const int width, const int height, const int stride,
   const float x, const float y)
2: {
3: float dx, dy, res, intPartX, intPartY;
4: int ix0, iy0, ix1, iy1;
5: dx = fabs(modf(x, &intPartX));
6: dy = fabs(modf(y, &intPartY));
7: ix0 = (int)intPartX;
8: iy0 = (int)intPartY;
9: if ix0 <0 then
10:    ix0 = abs(ix0 + 1);
11: end if
12: if iy0 <0 then
13:    iy0 = abs(iy0 + 1);
14: end if
15: if ix0 >= width then
16:    ix0 = width * 2 - ix0 - 1;
17: end if
18: if iy0 >= height then
19:    iy0 = height * 2 - iy0 - 1;
20: end if
21: ix1 = ix0 + 1;
22: iy1 = iy0 + 1;
23: if ix1 >= width then
24:    ix1 = width * 2 - ix1 - 1;
25: end if
26: if iy1 >= height then
27:    iy1 = height * 2 - iy1 - 1;
28: end if
29: res = t[ix0 + iy0 * stride] * (1.0f - dx) * (1.0f - dy);
30: res += t[ix1 + iy0 * stride] * dx * (1.0f - dy);
31: res += t[ix0 + iy1 * stride] * (1.0f - dx) * dy;
32: res += t[ix1 + iy1 * stride] * dx * dy;
33: return res;
34: }
```

Pragmáticamente el siguiente código es el responsable de crear una modificación temporal del segundo frame para que luego sea posible el cálculo de las derivadas de la imagen.

Algorithm 29 Bucle de operaciones de contorno.

```
1: __kernel void warpimage_CL(__global float *src, __global float *u, __global float
   *v, __global float *out, const int width, const int height, const int stride, const int
   offset)
2: {
3: int i, j, pos;
4: float x, y, tmp;
5: i = get_global_id(0);
6: j = get_global_id(1);
7: if i <height && j <width then
8:    pos = j + i * stride;
9:    x = (float)j + u[pos];
10:   y = (float)i + v[pos];
11:   out[pos] = tex2D_CL(src+offset, width, height, stride, x, y);
12: end if
13: }
```

Fijándonos en el código lo que se hace es muy sencillo, para cada *píxel* de la imagen lo que hacemos es sumar a su coordenada el vector de desplazamiento que tiene en ese momento, dándonos el *píxel* que estamos tratando desplazado. Con estos datos recogidos procedemos a llamar a la función explicada anteriormente para que nos devuelva el valor a asignar al *píxel* que estamos tratando.

En el siguiente código se realizan las derivadas espaciales y la derivada temporal.
Si miramos el código podemos observar que para cada *píxel* cogemos sus dos *píxeles* siguientes y anteriores en una una dimensión (i o j) y aplicamos el filtro $\frac{(1,-8,0,8,-1)}{12}$ sobre estos *píxeles*, una vez aplicado el filtro sobre estos, lo que se hace es realizar una media aritmética entre el valor obtenido en el frame siguiente y el anterior y ya hemos obtenido la derivada de esa dimensión. La derivada de i y j se calcula de la misma forma. Una vez que ya hemos computado los valores de Ix e Iy podemos computar Iz lo cual es tan solo hacer una resta

del frame siguiente menos el anterior y ya hemos obtenido las tres derivadas.

La función llamada en este código es similar a la explicada en el *kernel* de **downScale**.

Algorithm 30 Bucle de operaciones de contorno.

```
1: __kernel void computederivatives_CL(__global float *I0, __global float *I1, __global float *Ix, __global float *Iy, __global float *Iz, const int width, const int height, const int stride, const int offset)
2: {
3: int i, j, pos;
4: float t0, t1;
5: i = get_global_id(0);
6: j = get_global_id(1);
7: if i <height && j <width then
8:    pos = j + i * stride;
9:    t0 = tex2Di_CL(I0+offset, width, height, stride, j - 2, i);
10:   t0 -= tex2Di_CL(I0+offset, width, height, stride, j - 1, i) * 8.0f;
11:   t0 += tex2Di_CL(I0+offset, width, height, stride, j + 1, i) * 8.0f;
12:   t0 -= tex2Di_CL(I0+offset, width, height, stride, j + 2, i);
13:   t0 /= 12.0f;
14:   t1 = tex2Di_CL(I1, width, height, stride, j - 2, i);
15:   t1 -= tex2Di_CL(I1, width, height, stride, j - 1, i) * 8.0f;
16:   t1 += tex2Di_CL(I1, width, height, stride, j + 1, i) * 8.0f;
17:   t1 -= tex2Di_CL(I1, width, height, stride, j + 2, i);
18:   t1 /= 12.0f;
19:   Ix[pos] = (t0 + t1) * 0.5f;
20:   Iz[pos] = I1[pos] - I0[pos+offset];
21:   t0 = tex2Di_CL(I0+offset, width, height, stride, j, i - 2);
22:   t0 -= tex2Di_CL(I0+offset, width, height, stride, j, i - 1) * 8.0f;
23:   t0 += tex2Di_CL(I0+offset, width, height, stride, j, i + 1) * 8.0f;
24:   t0 -= tex2Di_CL(I0+offset, width, height, stride, j, i + 2);
25:   t0 /= 12.0f;
26:   t1 = tex2Di_CL(I1, width, height, stride, j, i - 2);
27:   t1 -= tex2Di_CL(I1, width, height, stride, j, i - 1) * 8.0f;
28:   t1 += tex2Di_CL(I1, width, height, stride, j, i + 1) * 8.0f;
29:   t1 -= tex2Di_CL(I1, width, height, stride, j, i + 2);
30:   t1 /= 12.0f;
31:   Iy[pos] = (t0 + t1) * 0.5f;
32: end if
33: }
```

El siguiente código desarrolla los cálculos (explicado en el capítulo anterior) para calcular du_{ij}^{n+1} y dv_{ij}^{n+1} mediante el método **Jacobi**.

Algorithm 31 Bucle de operaciones de contorno.

```
1: __kernel void solveforupdate_CL(__global float *du0, __global float *dv0, __global
   float *Ix,__global float *Iy, __global float *Iz, __global float *du1, __global float
   *dv1, const int width, const int height, const int stride, const float alpha)
2: {
3: int i, j, pos, left, right, up, down;
4: float sumU, sumV, frac;
5: i = get_global_id(0);
6: j = get_global_id(1);
7: if i <height && j <width then
8:    pos = j + i * stride;
9:    if j != 0 then
10:      left = pos - 1;
11:   else
12:      left = pos;
13:   end if
14:   if j != width - 1 then
15:      right = pos + 1;
16:   else
17:      right = pos;
18:   end if
19:   if i != 0 then
20:      down = pos - stride;
21:   else
22:      down = pos;
23:   end if
24:   if i != height - 1 then
25:      up = pos + stride;
26:   else
27:      up = pos;
28:   end if
29:   sumU = (du0[left] + du0[right] + du0[up] + du0[down]) * 0.25f;
30:   sumV = (dv0[left] + dv0[right] + dv0[up] + dv0[down]) * 0.25f;
31:   frac = (Ix[pos] * sumU + Iy[pos] * sumV + Iz[pos]) / (Ix[pos] * Ix[pos] + Iy[pos] *
      Iy[pos] + alpha);
32:   du1[pos] = sumU - Ix[pos] * frac;
33:   dv1[pos] = sumV - Iy[pos] * frac;
34: end if
35: }
```

4.3.3. Código OpenCL optimizado

Una vez ya teníamos el código anterior era hora de aplicar optimizaciones.

Las primeras tres optimizaciones se les considera como tal ya que aunque no sean cosas muy complejas si que son pequeñas mejoras que proporcionan unas mejoras de tiempo sustancialmente altas.

En primer lugar la primera optimización consiste en sustituir la necesidad de tener que leer datos del dispositivos en el *host* inicializarlos a 0 y volver a escribirlos en el dispositivo, lo cual surgía en partes del código en las cuales era necesario en cada iteración de un cierto bucle una inicialización de unas ciertas variables a 0, como por ejemplo la inicialización de las derivadas al comienzo de cada iteración del *warping*.

Algorithm 32 Bucle de operaciones de contorno.

```
1: __kernel void initialize_CL(__global float *src, const int width, const int height, const
   int stride)
2: {
3: int i, j;
4: i = get_global_id(0);
5: j = get_global_id(1);
6: if i <height && j <width then
7:    src[i * stride + j] = 0;
8: end if
9: }
```

En segundo lugar, tenemos la aceleración de otra pieza de la estructura del algoritmo de **Horn&Schunk** que producía latencia muy altas y cuya aceleración disminuye bastante los tiempos de ejecución. Se trata de la operación de intercambio usado principalmente para intercambios entre las variables de desplazamiento y sus derivadas. Con esta aceleración evitamos tener que leer de los dos *buffers* y tener que escribir ambos.

Algorithm 33 Bucle de operaciones de contorno.

```
1: __kernel void exchange_CL(__global float *A, __global float *B, const int width,
   const int height, const int stride)
2: {
3: int i, j;
4: float tmp;
5: i = get_global_id(0);
6: j = get_global_id(1);
7: if i <height && j <width then
8:    tmp = A[i * stride + j];
9:    A[i * stride + j] = B[i * stride + j];
10:   B[i * stride + j] = tmp;
11: end if
12: }
```

Mediante la última optimización menor conseguimos que al hacer la suma de u y v con $du0$ y $dv0$ nos ahorremos la necesidad de tener que leer de ambos *buffers* hacer la suma en el *host* y al final tener que volver a escribir en el *buffer* de la variable en la que tengamos que escribir el nuevo valor.

Algorithm 34 Bucle de operaciones de contorno.

```
1: __kernel void add_CL(__global float *src, __global float *u, const int width, const
   int height, const int stride)
2: {
3: int i, j;
4: i = get_global_id(0);
5: j = get_global__id(1);
6: if i <height && j <width then
7:    src[i * stride + j] += u[i * stride + j];
8: end if
9: }
```

Después de introducir estas pequeñas mejoras procederemos a presentar una mejora un poco más compleja que provoca también tiempos menores en el código, se trata de la **memoria local 2.0** que se trata de una nueva forma de cargar los datos en memoria local la cual explicaremos a continuación y que provoca menores tiempos en los *kernels* en los cual se aplica.

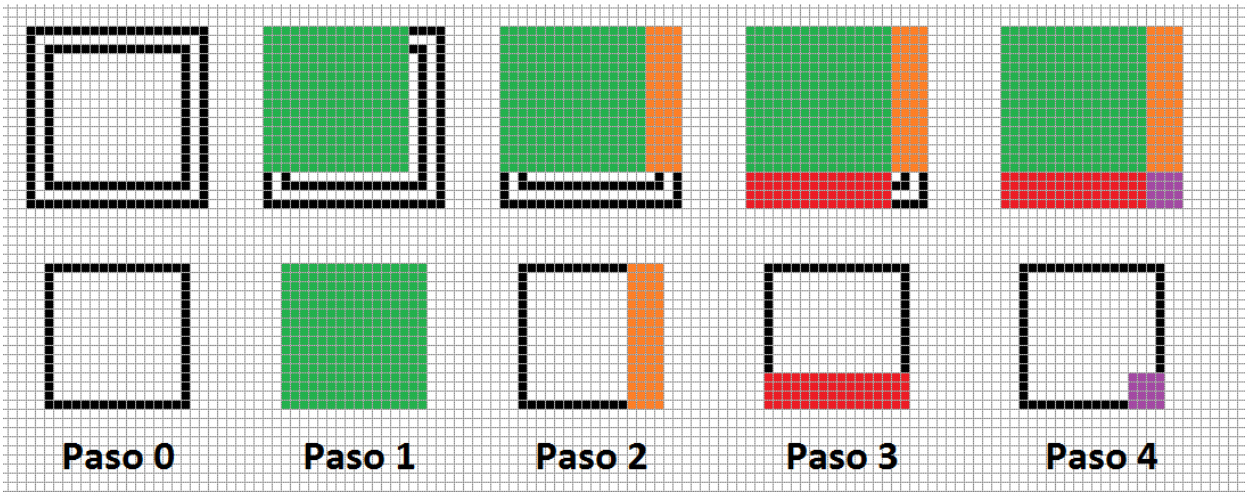

Figura 4.7: *Esquema de la memoria local 2.0*

Paso 0. Presentar un poquito este nuevo método de carga de información en la memoria local. Suponiendo que el *kernel* se ejecuta con un tamaño de *WorkGroup* de $16x16$ y el tamaño de la memoria local es de $20x20$ ya que sería los datos del bloque más los elementos que están a radio 2 de este bloque. Como se puede apreciar los elementos del bloque están centrados en la memoria local. Cada elemento tiene un identificador global (i, j) siendo i para las filas y la j para las columnas y uno local (li, lj) siendo li para filas y lj para columnas.

Paso 1. Todos los elementos del *WorkGroup* cargan el elemento que esta en memoria global en la posición $(i-2, j-2)$ en (li, lj).

Paso 2. Lo que hacemos es que las x últimas columnas, donde x es el doble del tamaño del radio, en este caso $2x2 = \mathbf{4}$ cargan las **4** últimas columnas que quedan por cargar en la memoria local para hacer que desde la fila 0 hasta la 16 esten cargadas. Es decir todos los elementos cuya $lj >= anchodelWorkGroup - \mathbf{4}$ y $j < anchodelproblema - 2$ cargan en $(li, lj + 4)$ los elementos globales $(i, j + 2)$.

Paso 3. Se realiza lo mismo que el paso anterior pero con las **4** últimas filas. Es decir

el **4** vuelve a ser por que es el doble del tamaño del radio, por lo que todos los elementos cuya $li >= alturadelWorkGroup - \mathbf{4}$ y $i < altodelproblema - 2$ cargan en $(li + 4, lj)$ los elementos globales $(i + 2, j)$.

Paso 4. Es la intersección de los dos pasos anteriores. El cuadrado de $4x4$ de la esquina inferior derecha (tamaño del radio x tamaño del radio) procede a cargar los datos que quedan por cargar. Es decir todos los elementos cuya $li >= alturadelWorkGroup - \mathbf{4}$, $i < altodelproblema - 2$, $lj >= anchodelWorkGroup - \mathbf{4}$ y $j < anchodelproblema - 2$ cargan en $(li + 4, lj + 4)$ los elementos globales $(i + 2, j + 2)$.

Para acabar con esta sección presentamos el código en **OpenCL** de esta nueva forma de carga en memoria local. Cuando una de las dimensiones de la imagen no es múltiplo de las dimensiones del bloque puede ser que se desajuste el relleno de datos en la memoria local y queden algunas posiciones sin inicializar. Para arreglar esto lo que hacemos es tener unas dimensiones virtuales del bloque las cuales modificamos solo si el tamaño útil del bloque (quitando *threads* que se salen de rango) es menor del tamaño real esto se representa por los dos primeros **ifs** que se pueden ver en el código. El código representa un cálculo donde el tamaño del radio era 1. La variable (i, j) son los identificadores globales del *thread*, $(local\_i, local\_j)$ son los identificadores locales, $(size\_local\_X, size\_local\_Y)$ son las variables que representan el tamaño virtual del *WorkGroup* y $(local\_x, local\_y)$ es la dimension del *WorkGroup*. Al final del código se pone una barrera para que haya sincronización entre *threads*.

Con este nuevo método tan solo sería incluir el algoritmo posteriormente mostrado en los *kernels* **computederivatives** y **solveforupdate** que son los únicos que pueden usar memoria local y tirar de la memoria local en lugar de la memoria global de los datos que queramos acelerar el acceso.

Algorithm 35 Bucle de operaciones de contorno.

```
1: if height - (group_i * local_x) <local_x then
2:    size_local_X = height - (group_i * local_x);
3: end if
4: if width - (group_j * local_y) <local_y then
5:    size_local_Y = width - (group_j * local_y);
6: end if
7: if i >0 && j >0 && i <height && j <width then
8:    //Paso 1
9:    local[local_i * (local_y + 2) + local_j] = du0[(i - 1) * stride + (j - 1)];
10:   //Paso 2
11:   if j <width - 1 && local_j >= size_local_Y - 2 then
12:      local[local_i * (local_y + 2) + (local_j + 2)] = du0[(i - 1) * stride + (j + 1)];
13:   end if
14:   //Paso 3
15:   if i <height - 1 && local_i >= size_local_X - 2 then
16:      local[(local_i + 2) * (local_y + 2) + local_j] = du0[(i + 1) * stride + (j - 1)];
17:   end if
18:   //Paso 4
19:   if j <width - 1 && i <height - 1 && local_i >= size_local_X - 2 && local_j >=
      size_local_Y - 2 then
20:      local[(local_i + 2) * (local_y + 2) + (local_j + 2)] = du0[(i + 1) * stride + (j +
         1)];
21:   end if
22: end if
23: //ESPERAMOS EL RELLENO DE LA MEMORIA LOCAL
24: barrier(CLK_LOCAL_MEM_FENCE);
```

Capítulo 5

Resultados obtenidos y observaciones

En está sección se presentarán los resultados obtenidos que consideremos de interés para el lector. Primero, asentaremos las bases del estudio así como el entorno de trabajo explicando de manera detallada el proceso para seguir con un estudio de la precisión del algoritmo en cuestión (este mismo proceso se hará para cada algoritmo). De esta manera, conseguiremos una configuración lo más óptima posible para continuar con el estudio de velocidad de ejecución y resoluciones que compararemos con otras implementaciones de referencia desarrolladas en **OpenACC** u **OpenMP**.

5.1. Material de trabajo

Debido al modelo de programación heterogéneo de OpenCL se han usado varios modelos de procesadores gráficos para realizar las pruebas y, así, hacer más tarde una comparativa. Entre las usadas encontramos: un acelerador **GPU Tesla K20c de NVIDIA**, con un procesador **Kepler GK110** y 2496 núcleos CUDA; una tarjeta gráfica **NVidia GeForce GTX 980**, con 2048 núcleos *CUDA* y una frecuencia de reloj de 1126 MHz; una **FPGA de Altera SDK for OpenCL**, con una placa de la familia S5-PCIe-HQ; un acelerador **Intel(R) Xeon Phi 31S1P** de 57 cores y una frecuencia de 1,1 GHz y el procesador **AMD A10-6800K APU with Radeon**, con una frecuencia de 4,10 GHz y 4 nucleos.

Se puede apreciar la diversidad de unidades de procesamiento con las que contamos, 2

GPUs, 1 procesador de propósito general y 1 acelerador, es decir, los tres tipos disponibles en el mercado que apoyan **OpenCL**, así nuestro estudio será más completo. Además, es necesario hacer una introducción a una nueva incorporación a lista de máquinas que permiten ejecutar **OpenCL**: una *FPGA*. Al ser algo reciente no hay muchos estudios sobre las ventajas de usar **OpenCL** en este contexto, algo en lo que nos centramos en la sección de trabajo futuro ya que este dispositivo destaca por su ratio rendimiento/consumo de potencia.

5.2. Estudio de precisión y mejor configuración

Ya se habló en la sección de introducción sobre las métricas y los estímulos que se iban a usar. A continuación, se mostrarán los resultados del estudio para Lucas&Kanade usando el estímulo de *DivergingTree* con las métricas de **Barron y Otte&Nagel**, por ser este uno de los estímulos de más uso en este campo. Además, se adjuntará una tabla con todos los estímulos usados para que el lector pueda ver, aunque sea en forma de tabla, todos los datos obtenidos.

Inicialmente se aborda un estudio de precisión numérica variando el tamaño de filtros especiales de **3, 5, 7, 9, 11 y 13** y, para cada uno de ellos, una ventana de vecindad de **3, 7, 15, 19, 23 y 31**. Ya que la precisión, por como es calculada, no es igual de precisa para los bordes, se hará también un estudio de la densidad; esto es, eliminando el error producido por los bordes para un tamaño de vecindad/filtro. De esta manera podemos ver el porcentaje de error que supone el cálculo de los bordes en modo de densidad; como ejemplo, para un tamaño de filtro de 7 y una resolución de 300x300 tendremos una densidad de $((300-7)^2)/300^2$ que resulta en una densidad del 90 %.

El motivo para realizar estas mediciones con las densidades es por la falta de información en los bordes de las imágenes, dependientes del tamaño de filtro y de la vecindad. Nos damos cuenta que conforme nos alejamos del borde encontramos una precisión mayor, lo que hace

que la precisión global aumente para los bordes. A partir de ahora hablaremos de error sin corrección (error al natural), error con corrección en filtro (error al retirar el filtro) y error con corrección en bloque (error al retirar el bloque).

El error calculado y mostrado en la figuras siguientes es el calculado como la media del error de todos los pixeles. Este error se calcula como la diferencia entre el error calculado y el error *ground-truth*. Se diría que el error mostrado es la media aritmética de la diferencia (por esta razón no es equivocado el pensar en un estudio de la densidad como se ha explicado anteriormente).

5.2.1. Estudio: Lucas&Kanade

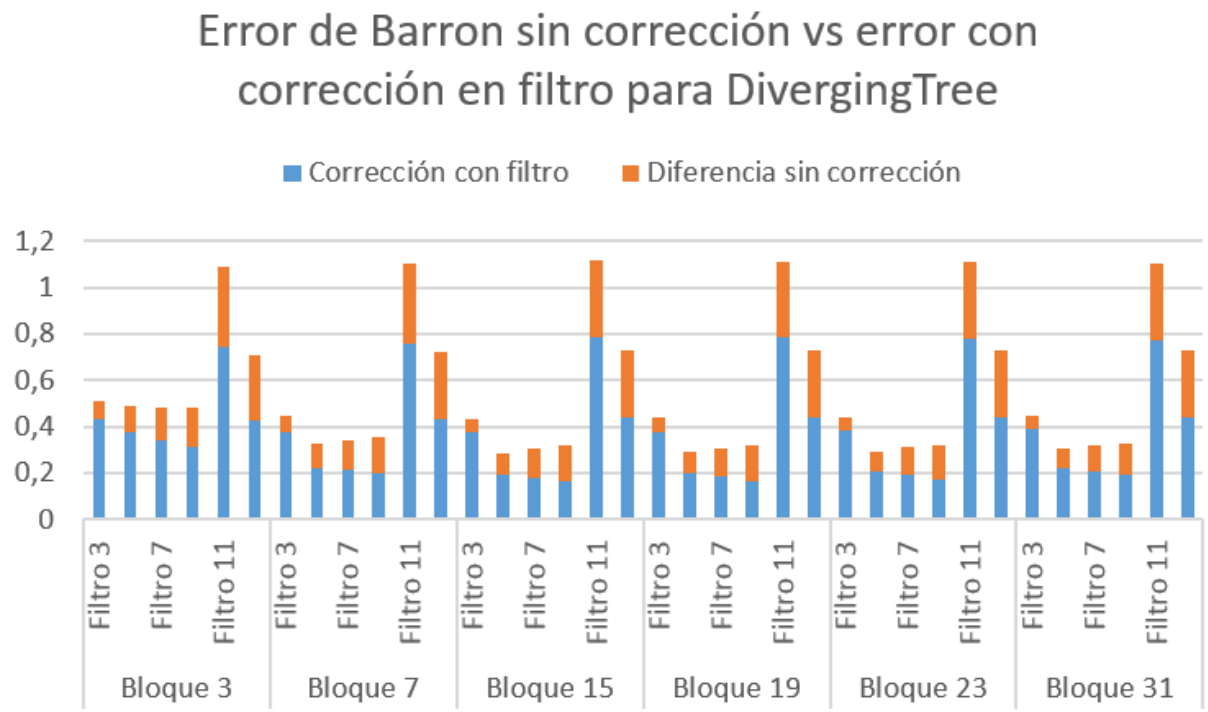

Figura 5.1: *Gráfica del error de Barron para DivergingTree con corrección en filtro.*

En la Figura 5.1 se puede ver como el mejor resultado en corrección de filtro lo ofrece el bloque 15 con filtro de tamaño 9 para el error de Barron.

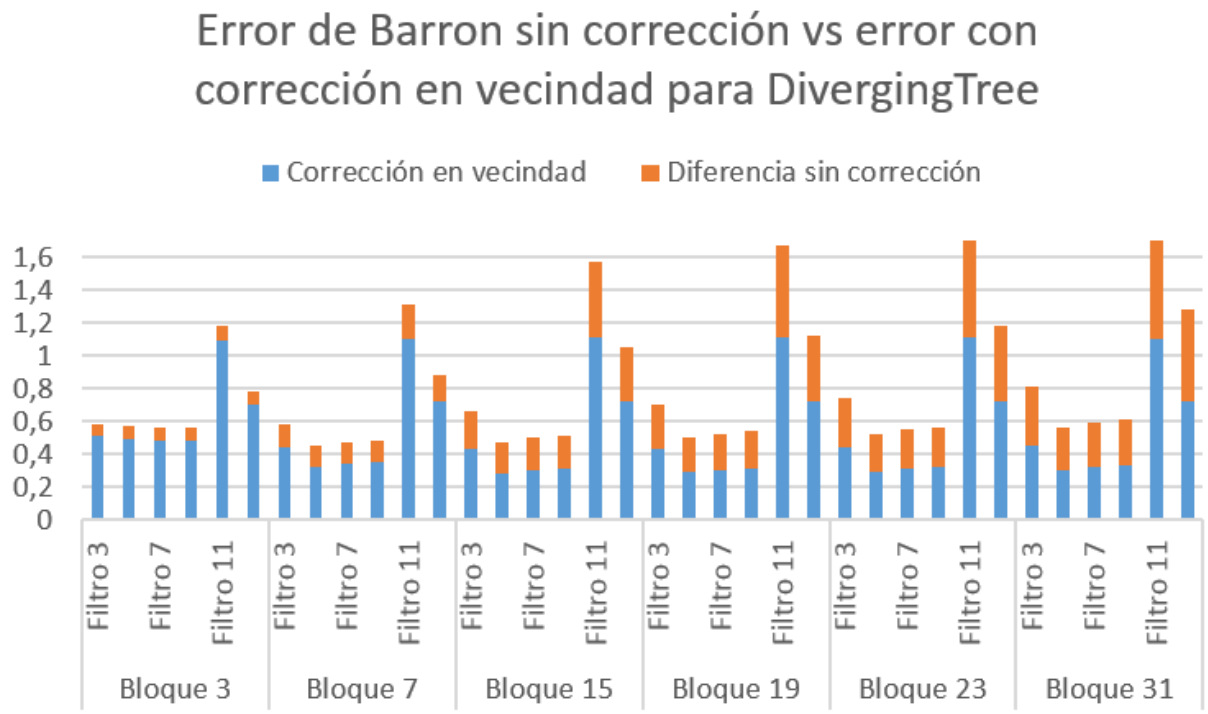

Figura 5.2: *Gráfica del error de Barron para DivergingTree con corrección en vecindad.*

Con la Figura 5.2 se observa como, obviamente, el error disminuye cuanto más grande es el bloque de vecindad que calcula el sumatorio, llegando a alcanzar los 0.04134 radianes de error Barron para una configuración de bloque 31 y filtro 5.

Con estos datos ya podemos obtener una comparativa por medio de una pequeña tabla para poner los datos sobre la mesa y, analizar así, la mejora que constituye la corrección.

Tipo	**Bloque**	**Filtro**	**Error Barron**	**Densidad**
Sin corrección	15	5	0,289	100 %
Corrección en filtro	15	9	0,16578	88,3 %
Corrección en bloque	31	5	0,04134	62,9 %

Cuadro 5.1: *Comparativa para los distintos tipo de corrección para el error de Barron.*

Gracias a la Tabla 5.1 se puede ver como varia el error para las distintas densidades para el error de Barron. Una simple cuenta nos dirá que al disminuir la densidad un 21,7 % se

consigue una mejora de 1,74 en el error y, al disminuirla un 37,1 %, se obtiene una mejora de 6,99.

El mismo estudio lo realizaremos sobre el error de Otte&Nagel. Compararemos las gráficas en corrección en filtro y en vecindad para localizar las mejores configuraciones. Una vez tengamos esto, se realizará una conclusión que desembocará en la configuración óptima cuya característica principal es que minimice la relación Mejora de error-Densidad sacrificada.

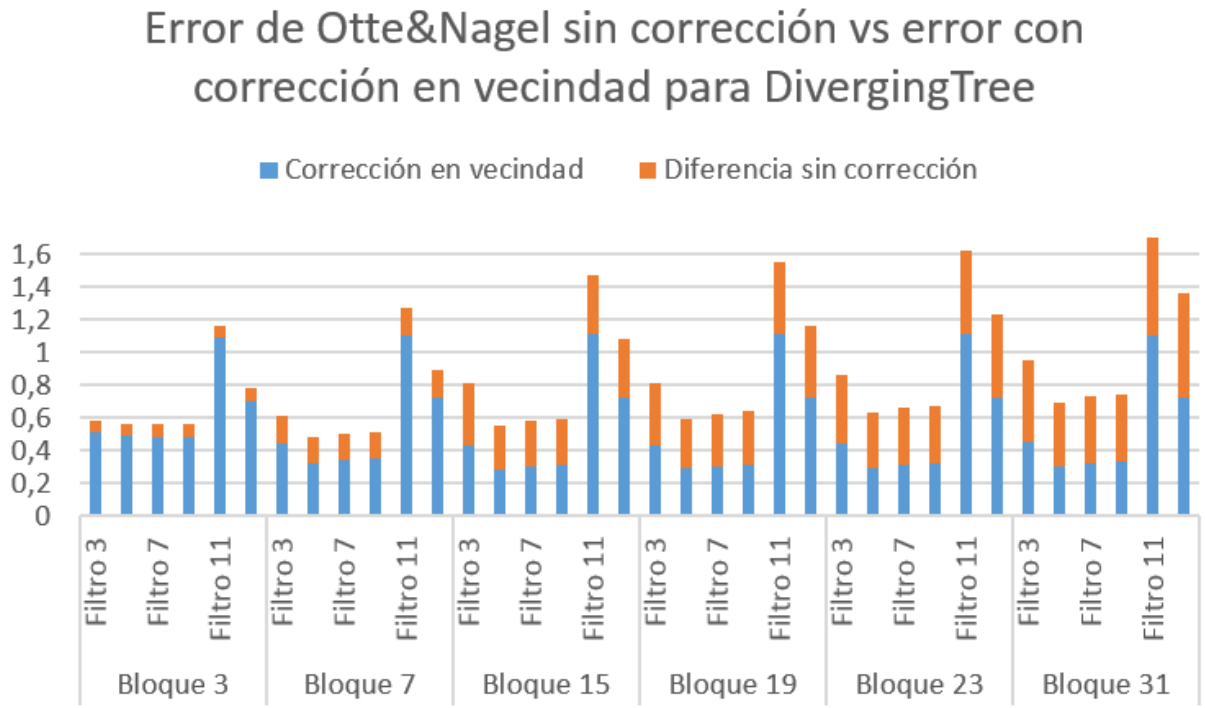

Figura 5.3: *Gráfica del error de Otte&Nagel para DivergingTree con corrección en filtro.*

En la Figura 5.3 podemos observar como se repite al igual que el error de Barron: la mejor configuración es bloque 15 y filtro 9. Nos gustaría destacar que configuración óptima depende del estímulo de entrada.

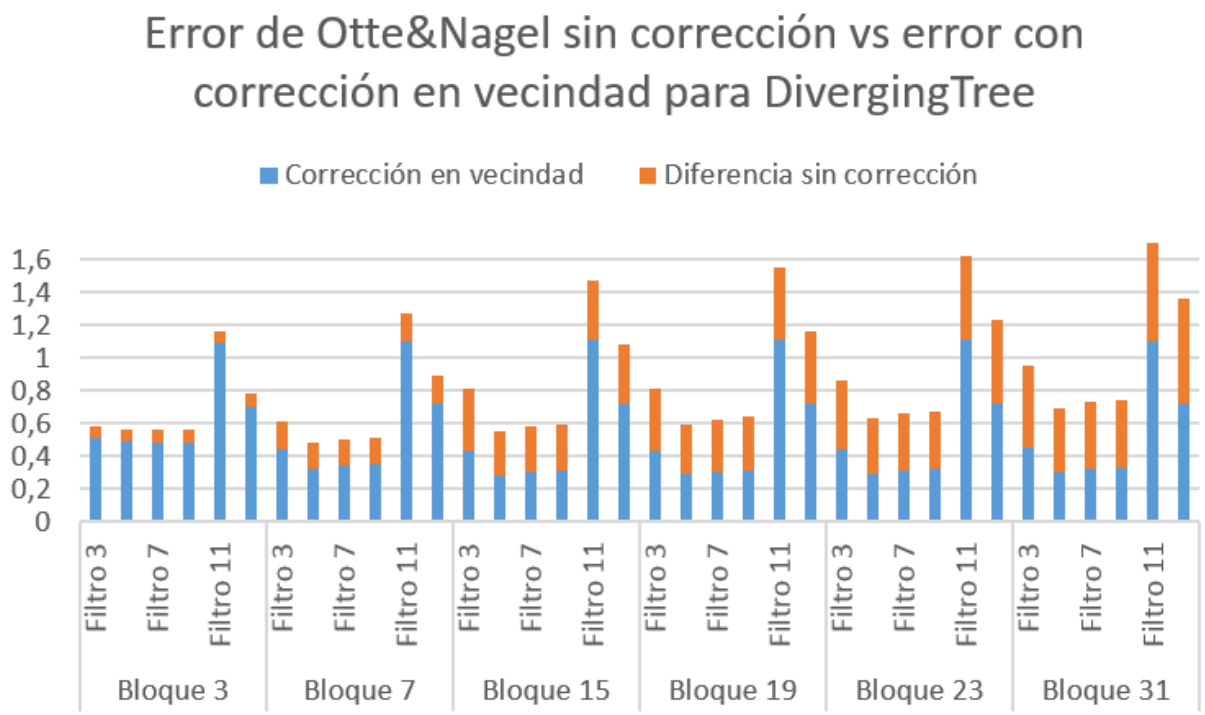

Figura 5.4: *Gráfica del error de Otte&Nagel para DivergingTree con corrección en vecindad.*

En este caso podemos observar que también se repite la configuración de bloque 31 y filtro 5 en la Figura 5.4. Con estos datos, podemos generar una tabla homologa a la anterior.

Tipo	Bloque	Filtro	Error Otte&Nagel	Densidad
Sin corrección	15	5	0,441165	100 %
Corrección en filtro	15	9	0,24561	88,3 %
Corrección en bloque	31	5	0,04939	62,9 %

Cuadro 5.2: *Comparativa para los distintos tipo de corrección para el error de Otte&Nagel.*

Gracias a todo el estudio realizado podemos concluir que la mejor configuración se encontraría en el bloque 15 y el filtro espacial de 5, al ser esta la configuración que mejores resultados da tanto en 100 % de densidad como en corrección por filtro y vecindad. Ofrece una variación de error alta sin comprometer demasiado la densidad calculada, que era el invariante que buscábamos minimizar.

A continuación, proponemos una tabla con más datos sobre otros estímulos para que el lector pueda ver el comportamiento del algoritmo Lucas&Kanade con otros datos de entrada.

	Errores		**Errores**		**Errores**	
Estímulo	**Barron**	**Otte&Nagel**	**Barron**	**Otte&Nagel**	**Barron**	**Otte&Nagel**
Grove2	1.33678	0.66198	1.41107	0.42520	1.37192	0.56525
Grove3	1.33335	0.65705	1.40595	0.43283	1.37052	0.56136
Urban2	1.33719	0.65878	1.40902	0.43359	1.43088	0.36936
TranslatingTree	1.44190	0.53603	1.45257	0.55921	1.06689	0.08579
Urban3	1.50173	0.77381	1.45444	0.73388	1.4847	0.74755

Cuadro 5.3: *Errores del algoritmo Lucas&Kanade para otros estímulos de entrada.*

En la Tabla 5.2.1 se han probado varias configuraciones: La primera columna de errores es para una configuración de bloque 15 y filtro 5. La segunda es para una configuración de bloque 31 y filtro 5, para ver como varía el tamaño del bloque de vecindad en el estímulo. Por último, la tercera es para una configuración de bloque 15 y filtro 13, para comprobar como vría el error conforme al tamaño del filtro espacial.

Queremos destacar en la Tabla 5.2.1 el hecho de que el error de Otte&Nagel es menor en todos los casos al de Barron, algo que no ocurría (como veíamos más arriba) con el caso de DivergingTree. Esto se explica con el tipo de estímulo que es generado: En DivergingTree es un estímulo de zoom, lo que quiere decir que el error de Otte&Nagel no será continuo por toda la imagen, si no que en los bordes será mayor que en el centro; mientras que en los demás casos son estimulos de movimiento con la cámara, lo que quiere decir que será un movimiento constante, algo muy positivo para el error de Otte&Nagel.

De hecho, destacamos el valor del error en TranslatingTree para una configuración de bloque 15 y filtro espacial 13. Al aumentar tanto el filtro conseguimos abarcar más pixeles en un mismo eje (ya sea X o Y). Por la naturaleza del video, un movimiento en el eje X, es lógico pensar que a mayor tamaño de filtro espacial mejor resultados aportará, sin llegar a

importar tanto el filtro en la vecindad.

5.2.2. Estudio general: Horn & Schunk

El algoritmo de Horn&Schunck es un algoritmo que puede llegar a obtener unos índices de error muy bajos, esto se debe a que es un algoritmo piramidal el cual se va moviendo por niveles refinando los vectores de desplazamiento. En este algoritmo hay 3 factores que intervienen en la generación de unos errores pequeños:

- **Número de niveles.**
- **Número de iteraciones de deformación.**
- **Número de generación de las derivadas.**

Normalmente el tercer factor suele valer 1.
Los otros factores no suelen tener un valor prefijado como el tercero, consiste en ir ajustando los parámetros dependiendo de si te interesa un error lo más bajo posible en cuyo caso se aumentarán ambas tanto como se desee puede ser que te interese un análisis en tiempo real en cuyo caso sería ponerles el mínimo valor o finalmente puede ser que te interese la mayor relación error/tiempo en cuyo caso la exploración de los mejores valores crece, ya que habría factores como por ejemplo hasta que nivel es rentable bajar sin perder rendimiento o cuando pasar de hacer cálculos en la GPU a hacerlos en la CPU por motivos como resoluciones muy bajas.

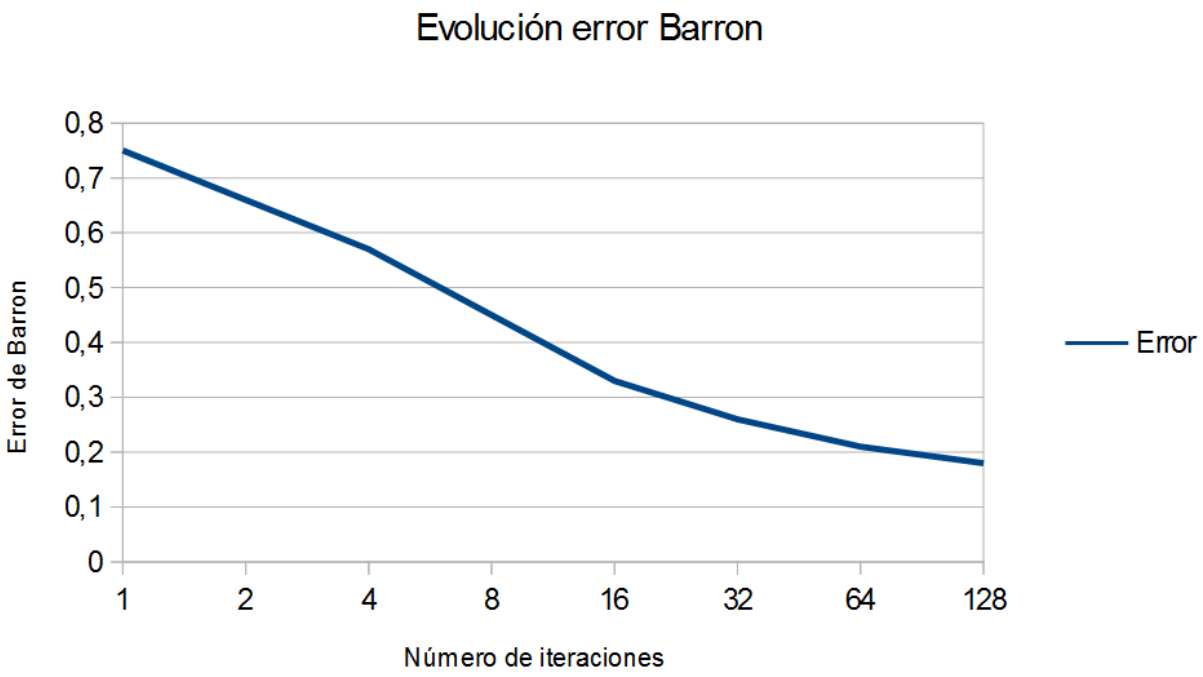

Figura 5.5: *Gráfica con la evolución del error con el estímulo de RubberWhale.*

Como se puede ver en la gráfica al aumentar el número de iteraciones el error disminuye considerablemente. Como se observa en la gráfica hasta un número de 32 iteraciones la reducción del error es bastante rentable siendo a partir de este valor cuando el grado de disminución del error se reduce notablemente de unos 0,09-0,12 a pasar a reducir tan solo 0,05 con 96 iteraciones más.

5.3. Rendimiento multiplataforma

5.3.1. Luka&Kanade

Como ya se discutió más arriba, vamos a continuar haciendo un estudio del tiempo aprovechando la heterogeneidad de OpenCL. Teniendo ya una configuración que consideramos óptima (al menos para el estimulo ejemplificado) vamos a comparar los distintos tiempos de ejecución que ofrecen los tipos de máquinas mencionados más arriba.

Para medir estos tiempos se ha considerado el estimulo DivergingTree con el que se hizo el estudio de precisión y la configuración de bloque 15 y filtro 5 para todas las máquinas. Esta ejecución nos servirá más tarde para ver el rendimiento obtenido mediante los FPS por cada plataforma.

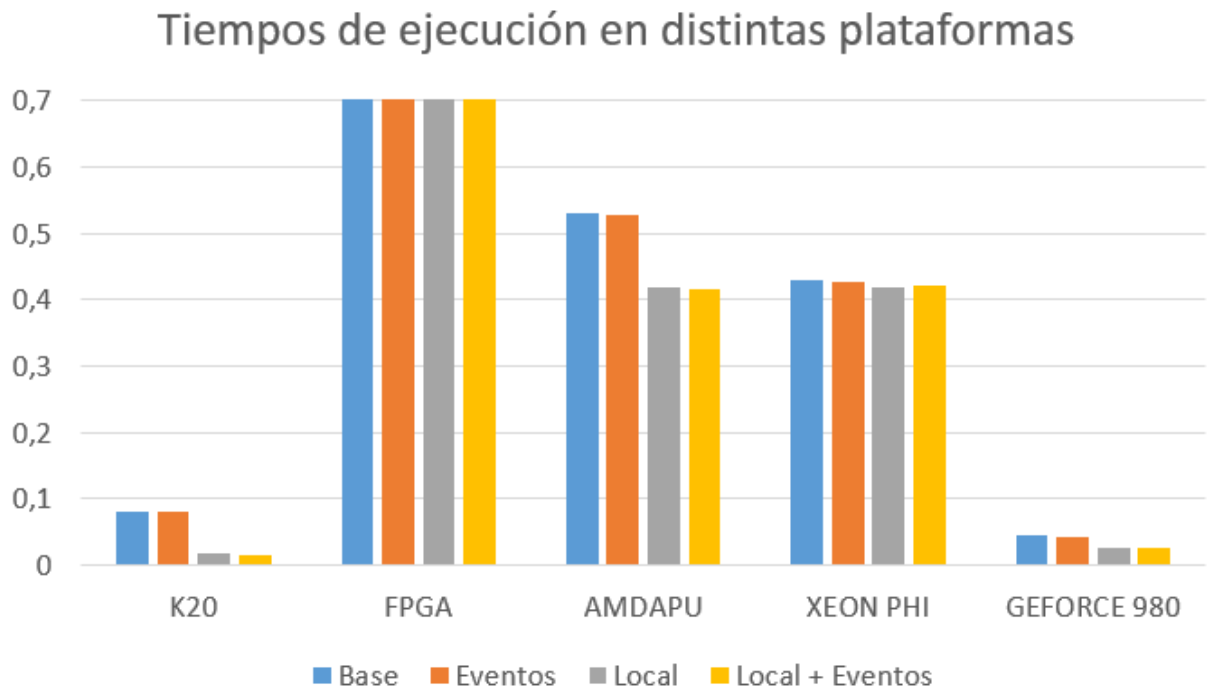

Figura 5.6: *Gráfica con el tiempo de ejecución de cada máquina con la configuración bloque 15 y filtro 5.*

En la Figura 5.9 se puede observar como las plataformas de **NVidia** ofrecen el mejor resultado, seguidas de cerca por las alternativas de **AMD** e **Intel** y en último puesto la *FPGA*. Es necesario especificar que esta comparativa no intenta sacar a la luz que gráfica es mejor, si ese fuese el punto, se escogerían gráficas de la misma generación. Con esta gráfica se busca que el lector vea que es posible la aceleración multiplataforma con **OpenCL** y que, además, los resultados obtenidos son tan buenos como permita la máquina. La linea que

cruza toda la gráfica es el tiempo que tarda el algoritmo en ejecutarse en modo secuencial.

Pero, pongamos un caso real: una cámara de seguridad está enviando información en forma de *frames* a una de estas máquinas. Esta, analiza la información y ofrece un resultado. Lo interesante sería que esta máquina diese el resultado en un tiempo casi instantáneo, si una persona pasa por delante de la cámara, que esta le siga conforme va pasando, no que empiece a seguirla cuando esta ya ha cruzado. Es por eso que es tan importante el tiempo real para este tipo de algoritmos.

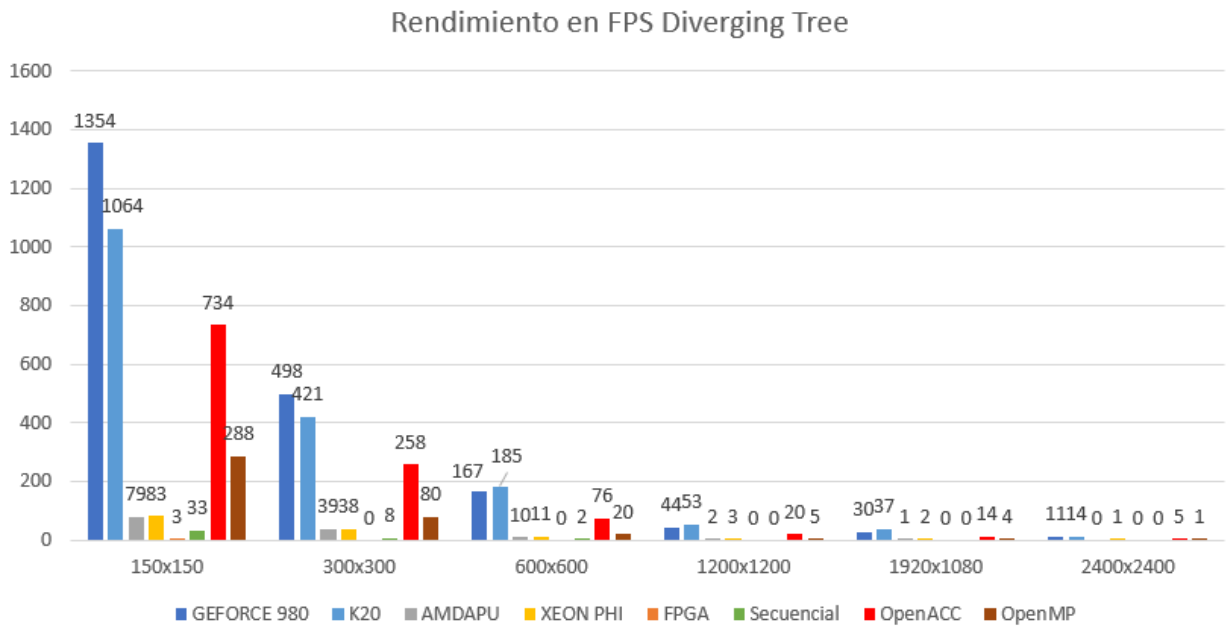

Figura 5.7: *Frames por segundos para la ejecución en distintas plataformas.*

Podemos considerar tiempo real que llegue a unos 24 *frames* por segundo. Con esto en mente, podemos ver en la Figura 5.7 como podemos llegar a obtener tiempo real con una resolución de $1920x1080$, la llamada resolución *FullHD*. Esta información es muy importante ya que: OpenCL ya no es solo un método de programación heterogéneo que permite ejecutar un mismo código sobre distintas plataformas, con estos nuevos datos sabemos que **ofrece mejor rendimiento** que lenguajes por medio de directivas[1, 17] como **OpenACC**

y **OpenMP**.

El llamado *Secuencial* ha sido medido sobre un procesador de propósito general y evoca al código en C original, por eso mismo no es raro ver como la **FPGA** es la única que se encuentra por debajo de este. También hay que tener en cuanta que **OpenACC** y **OpenMP** han sido lanzados en la máquina **Tesla K20c**.

Lo importante es destacar, como hemos hecho arriba, la mejora con respecto a otro tipo de programación: Estando en la misma plataforma, **OpenCL** es capaz de sacar un rendimiento 1.45 veces mayor a **OpenACC**, y no hablemos ya de **OpenMP**.

5.3.2. Horn & Schunk

Continuamos haciendo el estudio del tiempo en el algoritmo de **Horn&Schunk** y de como de buena es la la heterogeneidad de **OpenCL** sobre distintos dispositivos. Teniendo dos configuraciones que hemos escogido considerando diferentes necesidades, la configuración A de baja intensidad de carga con un error resultante de los peores que puede dar este algoritmo (como es obvio ya que se refina poco los vectores de desplazamiento) con un doble propósito el ver como evolucionan los tiempos de este algoritmo al meterle carga computacional y el poder comparar los tiempos con el algoritmo anterior en el peor de los casos y otra configuración B con más intensidad de carga y un error mucho menor. Vamos a comparar los distintos tiempos de ejecución que ofrecen los tipos de máquinas mencionados más arriba con ambas configuraciones de ejecución de este algoritmo.

Para medir estos tiempos se ha considerado el estimulo *Grove2* con el que se hizo el estudio de precisión y la búsqueda de las dos configuraciones usadas para todas las máquinas. Esta ejecución nos servirá más tarde para ver el rendimiento obtenido mediante los *FPS* en todas las plataformas.

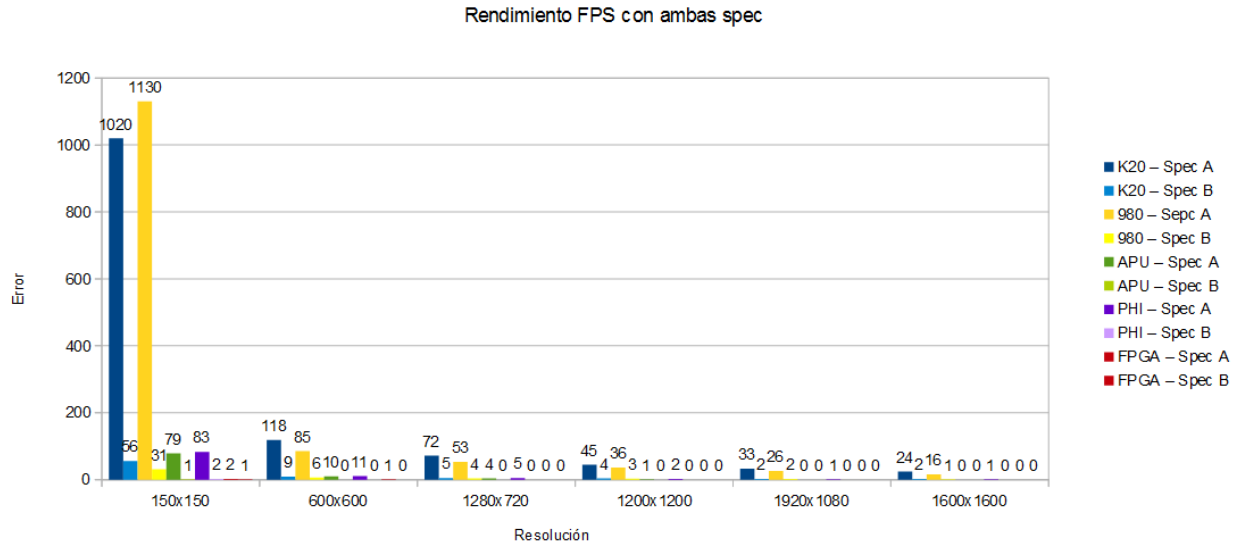

Figura 5.8: *Fps para las dos spec.*

En la Figura 5.8 se puede observar como con la configuración A las plataformas de **NVidia** ofrecen un resultado muy bueno incluso en resoluciones altas (la **K20** alcanza **tiempo real** incluso a $1600x1600$), las alternativas de **AMD** e **Intel** resisten como pueden el asalto en $150x150$ pero a partir de ahí caen ante el peso que tiene la carga de este algoritmo y en el último puesto se encuentra la *FPGA* de **ALTERA** la cual presenta comportamientos parecidos al presentado en el algoritmo anterior.

Para la configuración B ni siquiera las *GPUs* de **NVidia** aguantan más de un asalto, obteniendo tiempo real solo en $150x150$ y desplomandose en las siguientes resoluciones, el *hardware* de **Intel**, **AMD** y **ALTERA** caen ante el peso de esta configuración incluso en la resolución mas baja.

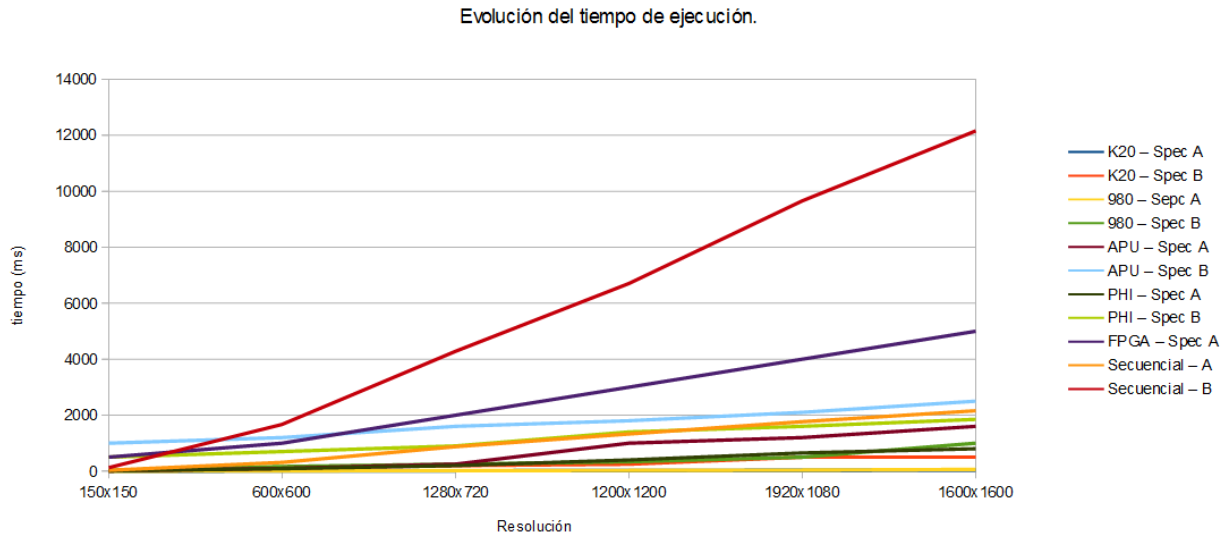

Figura 5.9: *Evolución de tiempos para las dos spec.*

Con esta gráfica se muestra que **OpenCL** es una gran solución para la aceleración de algoritmos multiplataforma y que, además, los resultados con una programación sin optimización específica de la plataforma se consigue muy buenos resultados. En la gráfica anterior se puede ver esto último comentado viendo que todos los tiempos incluso de máquinas que tienen una compatibilidad con **OpenCL** muy poco optimizada.

Tal y como dijimos en Luka&Kanade consideramos tiempo real si al menos llega a 24 *frames* por segundo. En este algoritmo alcanzamos **tiempo real** con una resolución de **1600x1600** con la K20 y en **1920x1080** con la **980** y **k20** lógicamente.

El llamado *Secuencial* como en el anterior algoritmo ha sido medido sobre un procesador **Intel i7 3770k** (propósito general) el cuál llama al código en C original, debido a esto la *FPGA* es la única que se encuentra cuyo tiempo de ejecución es superior al del código secuencial.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

Para realizar este Trabajo Fin de Grado ha sido necesario abordar preliminarmente un proceso de documentación sobre el concepto de estimación de movimiento y flujo óptico. Hay numerosas aplicaciones en este ámbito: Detección de enfermedades mortales procesando las imágenes que aportan las ecografías, *apps* dispositivos móviles que, haciendo uso de la cámara, son capaces de generar entornos virtuales, videovigilancia, procesado en entornos deportivos, control de calidad, codificación y transmisión de vídeo en *streaming* son potenciales aplicaciones que adquieren cada vez más relevancia en la era del **Internet of Things**.

Cabe destacar de forma adicional el gran interés que ha tenido siempre la comunidad científica para encontrar soluciones a problemas complejos computacionalmente de manera robusta[7], fiable y atendiendo a los requisitos de tiempo real que el problema demanda. El consumo de potencia es uno de los grandes aspectos a optimizar por parte de los dispositivos de cálculo siendo la búsqueda de compromiso entre exactitud, eficiencia y consumo de potencia un aspecto clave hoy en día. Creemos que la elección del paradigma de programación **OpenCL** puede ser la clave para este compromiso.

Se ha explicado a lo largo de la memoria, (sobre todo en capítulos iniciales) la pro-

blemática con los diferentes algoritmos de flujo óptico y las ventajas que tiene el uso de OpenCL respecto otros paradigmas de programación para *GPUs*. **Lucas&Kanade** junto a **Horn&Schunck** son dos grandes candidatos a conseguir una buena relación entre *accuracy-efficency*, de hecho, existen estudios que validan la estructura común de los algoritmos basados en gradiente[2]; fruto de esto es el modelo piramidal de **Lucas&Kanade**.

Con la cantidad de procesadores gráficos que hay actualmente en el mercado no es extraño que cada uno desarrolle su propio lenguaje de programación para poder ejecutar su código. Este hecho tienen como efecto la dificultad del reuso de código por parte de otras plataformas, teniendo que realizar modificaciones *ad-hoc* por parte del programador. Es por eso que el uso de **OpenCL** adquiere más importancia en este ámbito, pues las posibilidades de la heterogeneidad son notablemente superiores. A modo de ejemplo, no se necesita plataformas con una misma arquitectura para poder crear *clusters* de computación, se puede conformar uno heterogéneo si nos aseguramos que cada uno de los nodos tengan soporte **OpenCL**, independientemente del *performance* individual.

OpenCL supone una alternativa a considerar, en un mundo donde cada vez hay más dispositivos heterogéneos y se genera mayor cantidad de información. Pensemos que todos tenemos un supercomputador en nuestro bolsillo y realizamos cálculos difícilmente abordables por computadores más complejos hace unos pocos años. Con este proyecto se ha abordado un ejemplo de computación heterogénea en este nuevo ámbito, de manera competitiva y multiplataforma con soluciones extrapolables para otros dispositivos.

6.2. Trabajo futuro

Se plantea como idea inicial la optimización de los algoritmos usados en este trabajo. Existen modelos de algoritmos de gradiente en el que se combinan entrada con diferentes colores para realizar una computación multiespectral. Otra idea a desarrollar es fusionar

varios algoritmos [2], donde se combinan diferentes puntos fuertes de cada uno de ellos, minimizando sus debilidades individuales. A partir de esta filosofía, podemos rescatar el modelo piramidal de **Luka&Kanade**[1] (mencionado arriba).

La extensión de este trabajo al cálculo de disparidad binocular (estéreo)[8] y posteriormente el desarrollo de su estimación de movimiento en 3D es una de las interesantes líneas futuras a considerar. En este escenario se parte de la información tomada por dos pupilas de la misma escena y en el mismo instante (pero localizadas en distintos lugares del plano). Cada punto del mapa es proyectado a cada uno de las lentes, lo que genera dos proyecciones del mismo punto. Esto forma un entorno 3D en el que el objetivo es reconstruir la escena con la información individual de cada vista.

Debido a la heterogeneidad de **OpenCL**, el tipo de dispositivos con este soporte se incrementa cada vez más. En este mismo trabajo se presenta a las *FPGAs* como un plataforma viable para el uso de **OpenCL**. Es necesario, por tanto, hacer un estudio *accuracy-efficency* para diferentes tipos de datos, ya sean enteros o flotantes. Esta necesidad nace de la diferencia existente entre plataformas para llevar a cabo operaciones con tipos en coma flotante, por ejemplo, la **APU** de **AMD** tiene problemas para operar con valores de precisión *double* y la *FPGA* trabaja mucho mejor con valores enteros. Nuestros experimentos anteriores nos sugieren pensar en encontrar una configuración donde la precisión que sacrifiquemos sea coherente con el consumo en todo el espacio de soluciones. Por último, una de las ideas a considerar es evaluar este paradigma mediante medidas de complejidad del software [20] para intuir hasta que punto es viable tener a desarrolladores trabajando en un proyecto de **OpenCL**, donde lo usual es generar muchas líneas de código. Debido a los lenguajes basados en pragmas como **OpenACC** y **OpenMP**[1, 17] las lineas de código que se deben añadir han disminuido considerablemente.

[1]El flujo óptico es estimado por primera vez sobre la imagen de menor resolución, Esta imagen va creciendo en un factor X hasta alcanzar su tamaño inicial.

Aportación individual al proyecto

David Díaz Morgado

Durante el desarrollo de mi parte del TFG he realizado lecturas procedentes de *papers* y artículos académicos, además de diapositivas on-line y libros de texto para establecer un estudio sobre la teoría referente al algoritmo clásico de **Horn&Schunck** y a las métricas de error para aplicarlos sobre este algoritmo. También se ha hecho una búsqueda para encontrar diferentes implementaciones del algoritmo encontrando una en **C** y otra en **CUDA**.

Tuve que dedicar tiempo para tener un acercamiento a **OpenCL** para aprender el método de programación y la mejor manera para obtener resultados óptimos. Así empecé a desarrollar código para **OpenCL** a modo de prueba para acostumbrarme y empezar, así, con **Horn&Schunck**. También agradecer a mi compañero por enseñarme mediante la implementación de **Lucas&Kanade**.

A partir de este estudio previo OpenCL y de implementaciones secuenciales del mismo, puse en práctica todo lo que había aprendido con **OpenCL** paralelizando solo algunas funciones, mientras iba paralelizando estas funciones con **OpenCL** iba creando código *host* para hacer más reutilizable ciertas zonas del código, de manera que fuese más fácil el hacer pruebas y, a la par, reducir la cantidad de líneas.

Para continuar, realicé la paralelización del resto de código secuencial a **OpenCL**. Con todos los *kernels* necesarios en **OpenCL** una vez a mi disposición, volví a iterar sobre el código realizando las mismas tareas nombradas en el párrafo anterior (fomentar reutilización, facilitar pruebas, compactar código...). Para acabar, implementé optimizaciones para

acelerar aún más el código, estas optimizaciones consistían en:

- Adaptar al modelo de ejecución de **OpenCL** (anfitrión - dispositivo) a partes de esqueleto del **Horn&Schunck** secuencial totalmente válidas pero que hacían que el algoritmo ralentizase demasiado.
- Reducción de tiempos de acceso a memoria realizando en primer lugar un estudio de que *kernels* podrían hacer uso de la memoria local y si sería beneficiosa o no. Después de este estudio realicé la integración de la memoria local tal y como he comentado antes en esta memoria.

Una vez que ya todo el código estaba listo, me dispuse junto con mi compañero a sacar los tiempos de nuestros algoritmos para las diferentes máquinas. Se realizaron varios *scripts* en Octave que nos ayudasen a mecanizar un proceso de ejecución que podía llegar a tardar horas. Esta aproximación a Octave y a su lenguaje de *scripting* se hizo en conjunto.

La depuración de código y la solución de *bugs* corrió a cuenta propia de cada uno con el algoritmo que implementó. Por supuesto, no fue de manera aislada. Gracias al control de versiones de *git* podíamos ver que hacía cada uno cada vez que modificaba el código. De esta manera, cada uno de nosotros tenía control sobre lo que el otro hacía.

Después de obtener todos los datos y tener todos los tiempos y las gráficas de estos, además de todos los estudios de precisión, realizamos conjuntamente las partes comunes de la memoria y por separado las partes especificas de nuestros algoritmos. En particular, yo me dediqué sobre todo a la parte de *Hardware* y a todo lo referido a **Horn&Schunck**. Además completé varios puntos de la memoria mientras mi compañero iba corrigiendo errores para que la memoria quedase más legible.

David Gómez Blanco

La estructura ha sido la misma más o menos que la de mi compañero. Primero, para empezar con el TFG, se hizo una recopilación de información sobre los algoritmos. En mi caso, por medio de *papers* y artículos científicos de búsqueda propia o facilitados por los directores, hice un estudio de **Lucas&Kanade**, para entender como abordar este algoritmo y localizar los puntos clave de paralelismo.

A partir de otras implementaciones como la de C secuencial o las de **OpenACC/OpenMP** se desarrollo el código en **OpenCL**. No hizo falta una aproximación a **OpenCL** puesto que yo ya sabía codificar en este lenguaje. Por eso mismo, aproveche el tiempo en el que mi compañero estaba estudiando OpenCL para ayudarle mostrándole como codificaba **Lucas&Kanade**.

Primero, iba realizando los *devices* mientras continuaba con el código *host* de manera que cada vez que actualizaba los *devices* añadia la parte correspondiente en el *host* tal y como se especifica en la sección de **OpenCL** de la memoria.

Una vez hube probado el algoritmo sin optimización, pasé a realizar las mismas optimizaciones que las descritas por mi compañero. Cuando finalicé, mi compañero ya había terminado de estudiar **OpenCL** y estaba haciendo el algoritmo de **Horn&Schunck**, tiempo que aproveché para estudiar **OpenCL** para **FPGA** y así poder incluirla en el TFG.

Gracias a la documentación de ***ALTERA*** pude añadir las lineas de código necesarias para la ejecución de código **OpenCL** en una **FPGA** y compilar los *kernels* individualmente (junto con alguna modificación). Este trabajo realizado más tarde se lo explicaría a mi compañero para que lo pudiese añadir al código de **Horn&Schunck** sin necesidad de que hiciese un estudio previo.

Al terminar la codificación de sendos algoritmos, se pasó a realizar una captación de datos sobre el tiempo de ejecución, las métricas implementadas y los diferentes estímulos de entrada. Mediante *GNU Octave* mecanizamos todo este proceso y sacamos unas gráficas muy básicas que nos permitían ver si había algún error que teníamos que solucionar.

Mientras se solucionaban estos errores se empezaron a pasar los datos buenos a gráficas en *Excel* que resultaran más estéticas de cara a la memoria, que también se empezó a rellenar durante este proceso de prueba y error. Sobre todo, me encargué de una introducción a cada uno de los puntos, completando de manera general sobre lo que teníamos que hablar en la memoria, además de encargarme de todo lo relacionado con **Lucas&Kanade** y a las referencias y al apartado estético de la memoria, que engloba también revisión de fallos.

Añadir, que para hacer la memoria tuvimos que aprender *LaTex* que, aunque al principio tenga una leve curva de aprendizaje, resulta con él más sencillo hacer documentos y exportarlos a PDF. Además, unido al control de versiones de *git*, resultó más sencillo aún que el haberlo hecho en un archivo de *Word*.

Bibliografía

[1] Openacc Application Programming Interface. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf, 2013.

[2] Joachim Weickert Andres Bruhn and Christoph Schnorr. Lucas&kanade meets horn&schunck: Combining local and global optic flow methods. *International Journal of Computer Vision*, October 2005.

[3] El problema de la apertura. https://es.wikipedia.org/wiki/Introducci%C3%B3n_al_flujo_%C3%B3ptico#Problema_de_la_apertura, 2009.

[4] Fermin Ayuso. *Aceleración de algoritmos bioinspirados para estimación de movimiento en hardware paralelo*. PhD thesis, Universidad Complutense de Madrid, 2013.

[5] Simon Baker, Daniel Scharstein, J.P. Lewis, Stefan Roth, Michael J. Black, and Richard Szeliski. A Database and Evaluation Methodology for Optical Flow. *middlebury*, November 2011.

[6] J.L Barron, D.J. Fleet, and S.S. Beauchemin. Performance of optical flow techniques. *INTERNATIONAL JOURNAL OF COMPUTER VISION*, 12:43–77, 1994.

[7] G. Botella. *Implementación en hardware reconfigurable de un modelo de flujo óptico robusto*. PhD thesis, Universidad de Granada, 2007.

[8] Agustín Javier Salgado de la Nuez. *Métodos variacionales para la estimación del flujo óptico y mapas de disparidad*. PhD thesis, Universidad de Las Palmas de Gran Canaria, 2010.

[9] Javier Sánchez Enric Meinhardt-Llopis and Daniel Kondermann. Horn–Schunck Optical Flow with a Multi-Scale Strategy. *Image Processing On Line*, July 2013.

[10] Benedict R. Gaster, Lee Howes, David Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 225 WymanStreet, Waltham, MA 02451, USA, 1st edition, 2012.

[11] Berthold K.P. Horn and Brian G. Schunck. Determining optical flow. *Artificial Intelligence*, 1980.

[12] Jong-Nam Kim and Tae-Sun Choi. A fast three-step search algorithm with minimum chacking point using unimodal error surface assumption. *IEEE Transactions on Consumer Electrnics*, 44(3), August 1998.

[13] Kitti Vision Benchmark Suite A project of Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago. http://www.cvlibs.net/datasets/kitti/, 2009.

[14] Nicole S. Love and Chandrika Kamath. An empirical study of block matching techniques for the detecttion of moving objects. *Computer Vision*, 2006.

[15] B. Lucas and T. Kanade. An Iterative Image Registration Technique with an Application to Stereo Vision. *Artificial Intelligence*, pages 674–679, August 1981.

[16] B. McCane, K. Novins, D. Crannith, and B.Galvin. On Benchmarking Optical Flow. *Computer Vision*, August 2001.

[17] Openmp Application Programming Interface. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, 2013.

[18] M. Otte and H. H. Nagel. Estimation of Optical-Flow Based on Higher-Order Spatiotemporal Derivatives in Interlaced and Noninterlaced Image Sequences. *Artificial Intelligence*, 78(1-2):5–43, October 1995.

[19] Deqing Sun, Stefan Roth, and Michael J.Black. Secrets of Optical Flow Estimation

and Their Principles. *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, June 2010.

[20] Fernando Sáez Vacas. *Complejidad y Tecnologías de la Información*. Fundación Rogelio Segovia, Ciudad Universitaria, s/n, 28040-Madrid, 1st edition, 1990.